HIGH-PERFORMANCE MEMORY SYSTEM ARCHITECTURES USING DATA COMPRESSION

A Dissertation
Presented to
The Academic Faculty

By

Seungcheol Baek

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
in
Electrical and Computer Engineering

School of Electrical and Computer Engineering
Georgia Institute of Technology
May 2014

HIGH-PERFORMANCE MEMORY SYSTEM ARCHITECTURES USING DATA COMPRESSION

Approved by:

Dr. Jongman Kim, Advisor
Assistant Professor, School of ECE
Georgia Institute of Technology

Dr. Sudhakar Yalamanchili
Professor, School of ECE
Georgia Institute of Technology

Dr. David V. Anderson
Professor, School of ECE
Georgia Institute of Technology

Dr. Linda M. Wills
Associate Professor, School of ECE
Georgia Institute of Technology

Dr. Richard Vuduc
Associate Professor, School of CSE
Georgia Institute of Technology

Date Approved: March 2014

DEDICATION

To my parents

ACKNOWLEDGMENT

First and foremost, I would like to thank my advisor, Prof. Jongman Kim, for the optimism, enthusiasm, guidance, and encouragement during my graduate study at Georgia Institute of Technology. Without his continuous support, I could not have accomplished my Ph.D. study.

I would also like to express my gratitude to the thesis committee members, Prof. David V. Anderson, Prof. Sudhakar Yalamanchili, who also served on the reading committee, Dr. Linda M. Wills, and Dr. Richard Vuduc for their insightful and constructive comments and questions.

I would also like to extend my deep appreciation to Prof. Chrysostomos Nicopoulos and Prof. Hyung Gyu Lee. Their insight and advice were invaluable to my research and writing papers.

Thanks should also go to my lab mates, Dr. Junghee Lee, Suk Chan Kang and Vinson Young for being great friends to work with and for their helpful advice and reviews. I also very much appreciate supports from Jeomoh, Hak Jung, Jongkook, Jooncheol, Nak Seung, Young Bae, Min Su, Yaesuk, Daeyoung, Haejoon, Sangkil, and Taigon.

Last but not least, I am deeply indebted to my parents, Dae Hyun Baek and Young Ja Kim, for making all of this possible. Without their love and support, it would have been impossible to arrive at this point.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The seemingly irreversible shift toward multi-core microprocessor architectures has given rise to the chip multi-processor (CMP) archetype. This new paradigm has magnified the memory wall phenomenon, which is the result of the increasing performance gap between logic and off-chip memory devices. As a result, the need for a high-performance memory system has become imperative, since a well-designed memory hierarchy and sub-system is one of the most effective ways to bridge the logic-memory chasm. With burgeoning transistor integration densities [6], architects have partly responded to this challenge by dedicating increasing portions of the CPU's real estate to the Last-Level Cache (LLC). Even though modern architectures provide large LLCs to improve system performance, the size is not large enough to completely hide slow off-chip memory latencies because the working-set size of most applications tends to increase, over time, as well. Thus, effectively utilizing a given size of cache has been one of the most important research challenges so far in the field of microprocessor design.

The compression of LLCs is one of the most attractive solutions to cope with increasingly large working sets because storing compressed data in a cache increases the effective (logical) cache capacity, without physically increasing the cache size. Accordingly, this increased effective cache capacity can hold a larger working set and thereby improve the system performance significantly. This benefit has led researchers to develop various compressed LLC architectures by designing efficient compression algorithms [7, 8, 9, 10, 11, 12], or by architecting compression-aware cache structures to ease the allocation and management of variable-sized compressed cachelines[8, 12, 13, 5, 14, 15, 16, 17, 18]. However, no prior work has developed a cache management policy (mainly cacheline replacement policy) tailored to compressed caches. In other words, existing cache management policies for compressed caches employ the traditional cacheline replacement policies, whereby

only locality information is considered. Therefore, in this thesis, a compressed cacheline replacement policy that is aware of both the variable cacheline size information and the locality information is proposed.

Data compression helps memory-constrained applications, but, designed incorrectly, it can incur performance degradation because using data compression techniques has some overhead, such as compression/decompression latency, compaction overhead, and address remapping [7, 8, 9, 10, 11, 12, 13, 5, 14, 15, 16, 17]. Consequently, the fundamental challenge of using data compression is to avoid, or minimize, the overhead. In particular, the decompression latency for read hits degrades system performance significantly, because the decompression latency increases the memory access latency. Thus, to address the possibly debilitating issue of excessive read-hit decompression overhead in compressed LLCs, early decompression technique for frequently used compressed cachelines will be studied.

In an effort to address some of the shortcomings of dynamic random access memory (DRAM) technology as an off-chip device, several new memory technologies have been developed over the last few years. One of the most promising new actors is phase change memory (PCM), which is gaining a foothold in the research community, predominantly due to its attractive ability to scale very deeply into the low nanometer regime, low power consumption, non-volatility, and fast read performance [19]. However, there are some crippling limitations that prevent PCM from completely ousting DRAM from future systems: low write performance and limited long-term endurance. These drawbacks have led designers toward the adoption of various architectural techniques, which attempt to mitigate the deficiencies of PCM technology [20, 21, 22]. One promising such solution is the deployment of hybridized memory architectures that fuse DRAM and PCM, to combine the best attributes of each technology [23, 24, 25]. The conventional way to hybridize DRAM and PCM is by using the DRAM as an off-chip cache (i.e., an additional level in the memory hierarchy). Although this DRAM cache significantly reduces the number of PCM accesses, DRAM/PCM memory hybrids are still not enough by themselves to adequately address all

issues pertaining to the use of PCM. Thus, in this thesis, a dual-phase compression mechanism for DRAM/PCM hybrid environments and a multi-faceted wear-leveling technique for the long-term endurance of compressed PCM will be studied.

This thesis also includes a new compression-based hybrid multi-level cell (MLC)/single-level cell (SLC) PCM management technique that aims to combine the performance edge of SLCs with the higher capacity of MLCs in a hybrid environment. The storage density of PCM has been demonstrated to double through the employment of MLC PCM arrays [26, 27]. However, this increase in capacity comes at the expense of increased latency (both read and write) and decreased long-term endurance, as compared to the more conventional SLC PCM. These negative traits of MLCs detract from the potentially invaluable storage benefits. Therefore, to enjoy the vast capacity potential of MLC PCM, the compressed data size will be exploited to dynamically re-configure the memory space between MLC and SLC arrays so that PCM can provide SLC-like performance at MLC storage capacity.

CHAPTER 2

SIZE-AWARE CACHE MANAGEMENT FOR COMPRESSED CACHE ARCHITECTURES

The Chip Multi-Processor (CMP) paradigm has cemented itself as the archetypal philosophy of future microprocessor design. Rapidly diminishing technology feature sizes have enabled the integration of ever-increasing numbers of processing cores on a single chip die. This abundance of processing power has magnified the venerable processor-memory performance gap, which is known as the ”memory wall.” The logic-memory chasm is a limiting factor in overall system performance, thus necessitating the presence of a high-performance Last-Level Cache (LLC) as a mitigating factor. Indeed, a well-designed LLC is one of the most effective ways to hide the off-chip memory latency. Burgeoning transistor integration densities [6] have allowed architects to responded to this challenge by dedicating increasing portions of the CPU's real estate to the LLC. Such large LLCs are becoming common in the workstation and server segments. Even though modern architectures provide large LLCs to improve system performance, the size is not large enough to completely hide slow off-chip memory latencies, because the working-set size of most applications tends to increase over time, as well. Thus, one of the most important research challenges so far in the field of microprocessor design is how to effectively and efficiently utilize a given cache size.

Employing data *compression* within the LLC is one of the most attractive solutions to cope with increasingly larger working sets, because storing compressed data in a cache increases the effective (logical) cache capacity, without physically increasing the cache size. Accordingly, this increased effective cache capacity can hold a larger working set and thereby improve the system performance significantly. This benefit has led researchers to develop various compressed LLC architectures by designing efficient compression algorithms [7, 8, 9, 10, 11, 12], or by architecting compression-aware cache structures to

ease the allocation and management of variable-sized compressed cachelines [13, 5, 8, 14, 15, 16, 17, 12]. Most of the prior efforts, however, focused on minimizing the inherent deficiencies of compression-based schemes, such as compression/decompression latency, address remapping, and compaction overhead. While these studies have led to significant improvements and have enabled the efficient use of compressed LLCs, no prior work has developed a cache management policy (mainly cacheline replacement policy) tailored to compressed caches. In other words, existing cache management policies for compressed caches employ the traditional cacheline replacement policies, whereby only locality information is considered. However, our experiments in this work will clearly show that the conventional cache replacement policies suffer from a fundamental weakness when applied to compressed caches: they fail to account for the cachelines' variable size, and, thus, they do not fully utilize the benefits of compressed caching.

In sharp contrast to conventional cache architectures – where the cacheline size is constant – the physical size of a stored cacheline *after compression* varies according to the achieved compression ratio. This means that the eviction overhead (miss penalty) in a compressed LLC will also vary, based on the size of the evicted and evictee cachelines. This work will demonstrate that if the size information of the compressed cacheline is considered in the cache management process, the increase in the effective capacity of the compressed LLC will be maximized, while the eviction cost will be minimized. Section 2.2 will present a detailed motivational example to illustrate the benefits of size awareness. Hence, in addition to *locality* information, the *size* information of the cachelines should also be one of the prime determinants in managing the cachelines in compressed caches and in identifying appropriate eviction victims. In order to maximize the effective cache capacity and minimize the eviction overhead under compression, the two salient properties of size and locality must be appropriately combined, since they are inherently inter-twined.

The realization of the importance of cacheline size awareness constitutes the primary

motivation and driver of the presented work. This chapter proposes the notion of a *size-aware* cache management policy. The embodiment of this concept is the **Effective Capacity Maximizer (ECM)**, which is a mechanism targeting *compressed LLC architectures*. The ECM architecture revolves around four fundamental and closely intertwined policies: *Size-Aware Insertion (SAI)*, *Size-Aware hit Promotion (SAP)*, *Size-Aware Eviction Scheduling (SAES)*, and *Size-Aware Replacement (SAR)*. If the size of a compressed cacheline is larger than a pre-defined threshold, the SAI policy (i.e., insertion) gives said incoming cacheline higher priority to be evicted, while the SAP policy (i.e., promotion) dynamically manages the possibility of eviction (evaluated through a re-reference interval prediction value) by considering the size information when the cacheline is re-referenced (re-used). Finally, the SAES and SAR policies (i.e., eviction) select the victim cacheline within the eviction candidates, in order to minimize the eviction overhead and maximize the effective capacity. All the proposed cacheline management policies comprising the ECM mechanism are very lightweight, in terms of hardware implementation, because all techniques involved mostly re-use components already implemented in the existing compressed LLC infrastructure.

To validate the efficacy and efficiency of the proposed ECM mechanism, we perform extensive simulations using memory traces extracted from real multi-threaded workloads running on a full-system simulation framework. Specifically, simulations with a 2 MB compressed LLC configured as physically 4-way set associative, and logically up to 16-way, indicate that our ECM increases the effective cache capacity in a compressed LLC by an average of 19.4% and 23.9%, as compared to the Least-Recently Used (LRU) and Dynamic Re-Reference Interval Prediction (DRRIP) [1] policies, respectively. This effective capacity improvement increases cache performance by reducing the number of misses by an average of 11.3% over LRU and 5.6% over DRRIP. As a result, the ECM technique improves overall system performance (in terms of Instructions Per Cycle, IPC) by 8.7%

and 5.1%, as compared to a compressed LLC using the LRU and DRRIP replacement policies, respectively. Furthermore, detailed energy analysis indicates that ECM lowers energy consumption by 6.9% and 4.1%, respectively, over LRU and DRRIP.

The rest of the chapter is organized as follows: Section 2.1 provides background information and related work in LLC compression techniques. Section 2.2 presents a motivational example for the importance of cacheline size information in the cache management policy of compressed caches. Section 2.3 delves into the description, implementation, and analysis of the proposed ECM architecture. Section 2.4 describes the employed evaluation framework and presents the various experiments and accompanying analysis. Finally, Section 2.5 concludes the chapter.

2.1 Preamble and Related Work

2.1.1 Cache Compression Techniques and Replacement Policies

The benefits of data compression are easily observable in LLC memory architectures, where capacity is one of the most sensitive factors. As such, compression constitutes one of the most effective ways to increase the logical capacity of the memory system without increasing its physical capacity. There have been many studies on the employment of compression techniques within the LLC micro-architecture. So far, the focus has been on designing the compression-aware cache structure itself, rather than the compression algorithm. Existing approaches can be classified into two broad categories: (1) variable-segment caches, and (2) fixed-segment caches. Both design options try to optimize the cache architecture around the use of the employed compression algorithm.

The main drawback of variable-segment caches is the requirement of non-negligible space for remapping and compaction, which is pure overhead [5, 17]. Since the number of occupied segments by the compressed data is variable, depending on the compression ratio, the latter directly impacts the effective capacity. Due to this variability in occupied segments, variable-segment caches are very effective in maximizing the effective capacity of the cache. On the other hand, fixed-segment caches aim to exploit compression without

any severe cacheline compaction or manipulation overhead, by fixing the number of segments occupied by the compressed cacheline (usually up to 2 or 4) [13, 8, 16, 12]. Due to the fixed segment size, if the compressed data size is very small, some segments might have empty space, which is not available to other cachelines. Our experiments indicate that this situation happens frequently under most compression algorithms – irrespective of their complexity. Most studies exploit *zero-value* compression [7, 11, 12], which achieves high compressibility, but still produces a lot of unusable empty space within the cache. This implies that compression algorithms do not fully utilize the physical space of the cache, even when the compression scheme is highly efficient. Since our focus is on maximizing the effective capacity, as well as reducing the cache miss penalty, we will mainly exploit the *variable-segment* cache architecture for the rest of this chapter. However, it should be noted that the proposed ECM will also work in a *fixed-segment* cache architecture, because the underlying concepts are applicable to both design options.

While the cache size significantly affects system performance, the cacheline replacement policy (i.e., victim selection when cache misses occur) is also an important factor affecting overall performance. There has been extensive research in developing efficient cache management policies [1, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38]. Generally, it is accepted that the LRU replacement policy (and its approximations) behaves relatively well with most applications and access patterns (e.g., recency-friendly and sequential streaming). However, the LRU incurs performance degradation under thrashing and mixed-access patterns. To address these problems, one recent study proposed a Dynamic Re-Reference Interval Prediction (DRRIP)[1]. The DRRIP comprises two cache management policies: Bimodal RRIP (BRRIP) and Static RRIP (SRRIP). The BRRIP specifically addresses the performance of thrashing access patterns, while the SRRIP aims to improve the performance of mixed-access patterns. The performance of DRRIP is primarily shaped by how well it performs under the two aforementioned cache management policies, since it relies on a *Set Dueling* method [29] to select the best performing among the two policies.

Despite enhancing the cache performance well beyond the levels achieved by the conventional LRU policy – with low implementation cost – these advanced cache management algorithms still focus on exploiting locality (re-reference rate) information only. While this is perfectly adequate in conventional cache architectures, in compressed cache architectures, where the size of each cacheline is variable and a function of the compression ratio, the size information should also be considered, because it directly determines the effective cache size and the miss penalty. Nevertheless, no previous work attempted to develop a cache replacement policy targeting compressed caches. To the best of our knowledge, this is the first work to propose a compressed cache architecture augmented with a customized cache management policy that is aware of both the variable cacheline size information and the locality information. It will be demonstrated that this combined locality-and-size awareness yields much improved performance results.

2.1.2 Decoupled Variable-Segment Cache Architectures

A high-level, abstract view of a single set of a decoupled variable-segment cache architecture [5] with a 64B cacheline size is illustrated in Figure 1. While each cache set is physically 4-way set associative, logical 16-way set associativity can be achieved by using more tags alongside a variable-segmented data area. More specifically, each set of the cache is broken into 64 segments and the size of each segment is only 4 bytes (single-word). The effective capacity for a single set is given by:

$$\text{physically 4-way} \leq \text{effective capacity} \leq \text{logically 16-way}.$$

The *physically 4-way* term comes from the size of Data Area, while *logically 16-way* is achieved through the size of Tag Area. For instance, if there are only uncompressed cachelines, the cache would operate like a typical 4-way set associative cache. Conversely, if there are only highly compressed cachelines, the cache would operate with 16-way set associativity. Therefore, each set can potentially increase its effective capacity by up to four times (when storing 16 compressed cachelines). Of course, one could have more than 16

cachelines in a set with larger tag space. The scalability of the mechanism proposed in this work with the number of logical ways will be explored in Section 2.4.4. Data segments are stored contiguously in Address Tag order. The offset for the first data segment of cacheline k (in a particular set) is

$$\text{segment\_offset}(k) = \sum_{i=0}^{k-1} \text{actual\_size}(i).$$

The *Cacheline size* tag in the "Tag Area" of Figure 1 is used to record the actual physical size of each cacheline (i.e., number of segments) in each set of the compressed LLC.

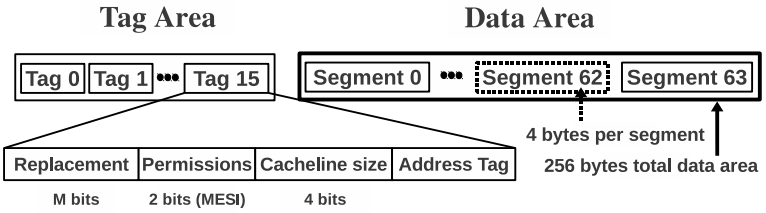


Figure 1: One set of a decoupled variable-segment cache.

2.2 Importance of Size-Aware Management

2.2.1 The Motivation for Size Awareness

The cache replacement policies in conventional cache architectures are optimized to minimize the off-chip memory accesses by considering the data access patterns during execution. However, as previously mentioned, in a compressed cache, the size information of the cacheline is equally important and should be exploited in the optimization of both the cache structure itself and the cache replacement policy. While the cacheline size is a constant parameter in conventional caches, it becomes a performance-critical variable under data compression.

Figure 2 shows the behavior of a compressed cache under the LRU replacement policy and a size-aware replacement policy. In this example, the cache is physically configured as a 2-way set associative cache, but logically configured as an 8-way set associative decoupled variable-segment cache. When U2 – a new cacheline – is requested, a cache miss

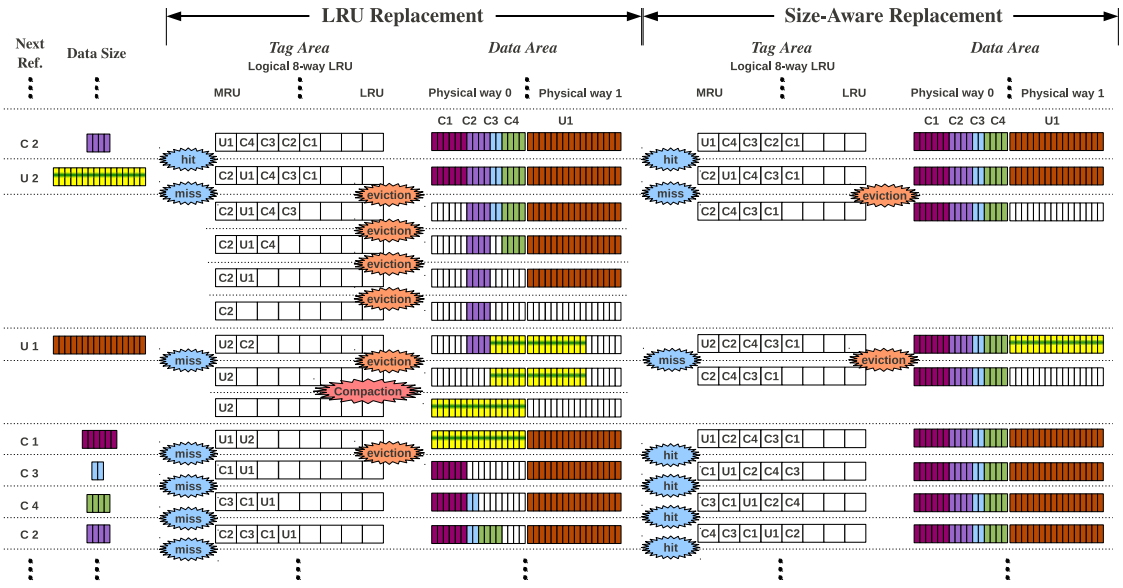


Figure 2: An example comparing the behaviors of the LRU replacement policy and a size-aware replacement policy in a compressed decoupled variable-segment cache. The cache is physically 2-way and logically 8-way set associative.

occurs. Four cachelines (C1, C3, C4, and U1) should be evicted under the LRU replacement policy. These evictions will lead to an almost *four-fold* increase in the miss penalty, as compared to the penalty of a conventional cache without compression. After the evictions of said 4 cachelines, C2 will also be evicted when the second U1 request arrives. In addition, a compaction process to allocate U1 should be performed in the data area, which results in non-negligible overhead in decoupled variable-segment caches. More importantly, at this moment, the Data Area contains only 2 cachelines, which is the minimum of its potential effective capacity. Even worse, the cache will experience 4 consecutive misses in the next 4 requests (C1, C3, C4, and C2). Hence, in this conventional configuration, a total of 6 cache misses and 6 cacheline evictions are observed, and the average number of cachelines that the cache set holds is 3.29.

On the other hand, we can vastly improve this situation by considering the *size* information of the evicted cacheline. As opposed to the previous example, we evict only U1 – which is the largest-size cacheline in the Data Area – when U2 first arrives. C1, C3, C4, and C2 can stay in the cache. Even though U2 will again be replaced by U1 at the next

request for U1, the following four requests (C1, C3, C4, and C2) after U1 will hit in the cache. Thus, in total, only 2 cache misses and 2 cacheline evictions are observed, and the average number of cachelines in a cache set increases to 5.0 in this configuration.

This example emphatically demonstrates that it is imperative to consider the variable cacheline size in compressed caches, in order to maximize the performance enhancement afforded by the compression scheme.

2.2.2 The Potential Conflicts Between Locality and Size Information

Having established the importance of both the *locality* and the cacheline *size* attributes in managing compressed LLCs, it is clear that the main challenge in this work is the effective *simultaneous* consideration of these two pieces of information, in order to maximize the benefits afforded by compression. Even though the locality and size information of a certain cacheline can vary over the execution timeline, one can simply classify a cacheline at a given time, *t*, into four possible status types: (1) high locality and small size, (2) high locality and big size (almost uncompressed), (3) low locality and small size, and (4) low locality and big size. For types (1) and (4), the decision is clear: keep (1) as long as possible, while evict (4) as soon as possible. If the biggest portion of the application favors these two types, the size-aware policy can easily be applied. On the other hand, for types (2) and (3), where the size and locality information are conflicting, a more complicated decision scheme is needed. Otherwise, the cache will experience performance degradation, even if the effective capacity increases through the consideration of size information. The solution for type (3) can still be relatively simple. We can evict this type of cacheline without any severe consequences, because the possibility of reusing it is low. However, the solution for type (2) is not easy, because the possibility of reusing such lines is high, but they occupy many segments (large space) in the cache. In order to consider this problematic case, a threshold-based classification process will be performed. Based on the classification result, each cacheline will be treated differently during cache insertion.

The relationship between the locality and size information of real application workloads

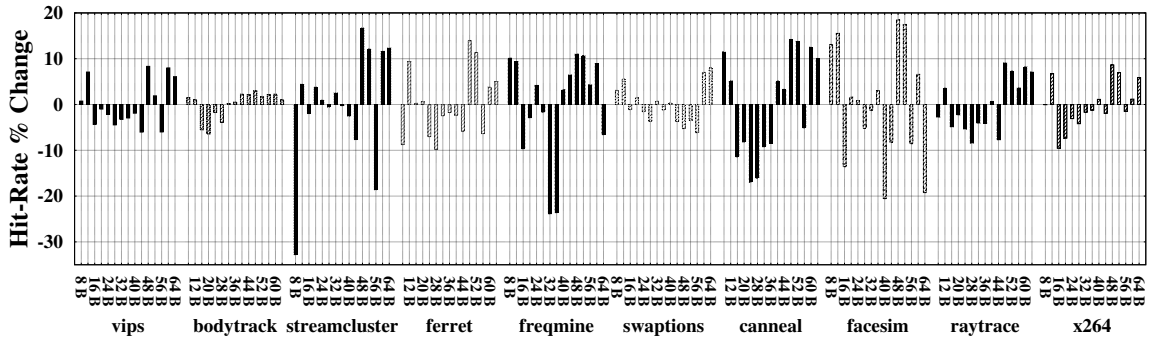


Figure 3: Hit-rate percentage change for each cacheline size over several PARSEC benchmarks [2].

is first analyzed, in order to identify which types of cachelines are more dominant over others in real-world scenarios. Several applications selected from the PARSEC benchmark suite [2] are executed under the assumption of a compressed cache using a 3-bit DRRIP[1]. The details of our evaluation framework are presented in Section 2.4. Figure 3 shows the hit rate variation, based on the cacheline size (in bytes). We measure the hit rate separately by cacheline size and normalize the values to the average hit rate of the cache. So, if the bar is above zero, a larger number of hits (higher locality) is observed for that specific cacheline size. Instead, if a bar is below zero, fewer hits (low locality) are observed. Depending on the application, high locality is observed both in small-size cachelines – type (1) – and in big-size cachelines – type (2). Further, type (3) and (4) cases are observed throughout the graph as well. This means that there is no clear relationship between the size and the locality, but a non-negligible number of type (2) cases (high locality and big size) are observed when the cacheline size is more than 48 B in most applications. Hence, our policy aims to maximize the benefits even under these conflicting cases. Our exploration with the entire PARSEC benchmark suite [2] indicates that significant numbers of such conflicting cases are invariably present in all applications.

[1]In this chapter, we use 32-entry *Set Dueling Monitors*, a 10-bit single-policy selection (PSEL) counter, and ϵ=1/3 for DRRIP [39, 1, 29].

2.3 The Effective Capacity Maximizer (ECM) Scheme

The exploitation of cacheline size information when selecting a victim in compressed LLCs is a necessary condition to maximize the benefits of compression; namely, enhancing the effective capacity – which implies a reduction in the number of misses – and reducing the miss penalty. In this section, we propose a size- *and* locality- aware compressed cache management scheme, called the Effective Capacity Maximizer (ECM), to further increase the performance of compressed LLCs. ECM's operation revolves around four policies: Size-Aware Insertion (SAI), Size-Aware hit Promotion (SAP), Size-Aware Eviction Scheduling (SAES), and Size-Aware Replacement (SAR). These basic policies can be used in conjunction with most existing locality-aware replacement policies. However, in this work, we partially exploit the basic framework of RRIP [1], which achieves high performance at a fairly low implementation cost, as compared to the other conventional cache management implementations. In fact, the RRIP framework already provides most of the information required by ECM, thus making the additional overhead due to the proposed scheme near-negligible.

Figure 4 illustrates the fundamental components of the ECM framework. The main operation of the proposed new cache management mechanism can be decomposed into three distinct phases (grey boxes in Figure 4), which include the four size-aware policies introduced in this section: (1) insertion (SAI), (2) promotion/demotion (SAP), and (3) eviction (SAES+SAR). This framework is inspired by the RRIP implementation [1] and it directly exploits the operational principles of RRIP when considering the locality attributes of cachelines. However, ECM goes beyond the basic RRIP premise with significant – yet lightweight – modifications/augmentations that enable the simultaneous consideration of locality and size information. Unlike RRIP, in each of the three main phases of the ECM mechanism, the size information of the cacheline is an integral part of the management decision.

Before delving into the details of each of the major components of the proposed ECM

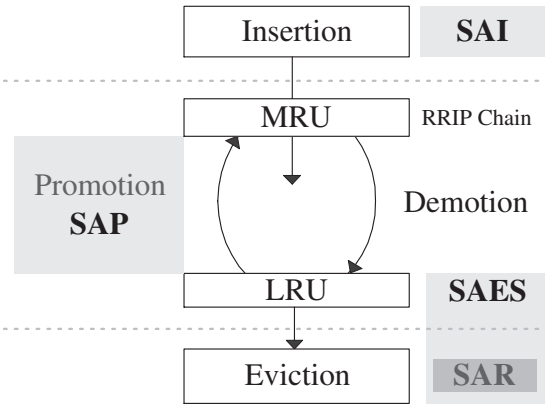


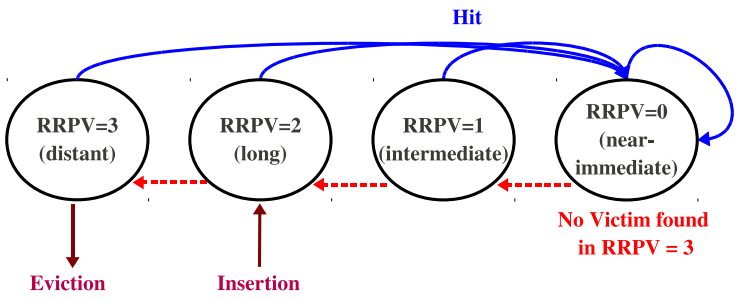
Figure 4: A high-level overview of the proposed ECM architecture.

architecture, we present the fundamental principles of RRIP [1], which form the foundations on which ECM is built.

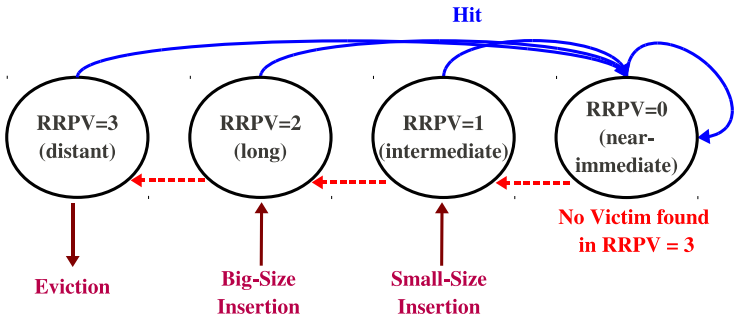
2.3.1 The Fundamental Principles of the Re-Reference Interval Prediction (RRIP) Framework [1]

Figure 5(a) shows the RRIP [1] framework, which uses M bits of meta-data for each cacheline, in order to store one of 2^M possible *Re-Reference Prediction Values* (RRPVs). The four possible RRPV values form an RRIP chain (one for each cacheline in the cache), as depicted in Figure 5(a). If a cacheline stores an RRPV of zero, it is predicted to be re-referenced (re-used) in the *near-immediate* future. On the other hand, if a cacheline stores an RRPV of $2^M - 1$, the cacheline is predicted to be re-referenced in the *distant* future. In other words, cachelines with smaller RRPVs are expected to be re-referenced sooner than cachelines with larger RRPVs. Therefore, on a cache miss, RRIP selects a victim among the cachelines whose RRPV is $2^M - 1$ (*distant* re-reference interval). If there is no cacheline with an RRPV of $2^M - 1$, the RRPVs of all cachelines are increased by 1 – called *demotion* – and this increasing process is repeated until a victim cacheline is found. On a hit, the RRIP changes the RRPV of the hit cacheline to zero – called *promotion* – thus predicting the cacheline to be re-referenced in the *near-immediate* future. When new data is initially fetched from the memory device into a specific cacheline, the Static RRIP (SRRIP) policy sets the cacheline's initial RRPV as $2^M - 2$, which indicates *long* re-reference interval,

instead of *distant* re-reference interval, to allow SRRIP more time to learn and improve the re-reference prediction. However, when the re-reference interval of all the cachelines is larger than the available cache, SRRIP causes cache thrashing with no hits. To address such scenarios, Bimodal RRIP (BRRIP) sets the majority of new cachelines' initial RRPVs as $2^M - 1$ (*distant* re-reference interval prediction), and it infrequently inserts new cachelines with RRPV of $2^M - 2$ (*long* re-reference interval prediction). Dynamic RRIP (DRRIP) determines which policy is best suited for an application between SRRIP and BRRIP using *Set Dueling* [29], which is a method that simultaneously monitors the performance of both SRRIP and BRRIP and chooses the winner among the two.



(a) RRIP framework



(b) SAI framework

Figure 5: The meta-data chains of the RRIP [1] and the proposed SAI policies, assuming the use of a 2-bit (M=2) RRPV. Note that each cacheline in the cache has its own RRPV value, which implies that each cacheline traverses its own chain, based on the line's hit/miss performance.

2.3.2 The Size-Aware Insertion (SAI) Policy

The insertion policy of a cache management framework decides where to insert an incoming cacheline in the meta-data chain. Specifically, in the case of RRIP, the insertion policy decided the RRPV value to be given to new cachelines. Whereas RRIP makes this decision based solely on locality (re-reference rate) information, the proposed SAI policy of ECM must also account for the incoming line's size. The main goal of the SAI policy is to give the big-size cachelines a higher chance of *eviction*, while minimizing the *conflict with locality*. For this purpose, we simply classify the cachelines into two categories: big-size cachelines and small-size cachelines. We then adjust the initial insertion point of the cacheline's re-reference interval, based on this classification. The cachelines classified as big-size are allocated with a higher RRPV than the small-size cachelines, so as to force the big-size cachelines to be evicted sooner. However, if the RRPV of big-size cachelines is set to $2^M - 1$ (*distant* re-reference interval prediction), the big-size cachelines with high locality will also be evicted sooner, due to the lack of time to learn locality information. Therefore, the RRPV of big-size cachelines is set to $2^M - 2$ (*long* re-reference interval prediction) and that of small-size cachelines to $2^M - 3$ (*intermediate* re-reference interval prediction), so that all the cachelines have enough time to learn locality information. Figure 5(b) shows the basic operation of the proposed SAI policy. Since only the initial insertion point (RRPV) in the RRIP chain must be modified, as compared to the baseline RRIP mechanism, no significant hardware alterations are necessary to support the SAI scheme. However, the critical issue in this methodology is how to classify a cacheline as either big-size, or small-size. This classification is very important, in order to balance the big-size cacheline thrashing effect and the re-reference interval prediction.

2.3.2.1 The Static Threshold Scheme

One simple policy is to use a predefined threshold, T_h ($2 \leq T_h < 16$, where 16 is equal to the size of a single uncompressed cacheline in number of segments, and 2 is the number of segments occupied by the smallest possible compressed cacheline). Each segment is 4 B

long, and a threshold is defined, in terms of a number of segments. Adjusting the threshold value translates to adjusting the balance between the locality and the size information. If the threshold is set high, the re-reference rate will be given more weight than the size information, while a lower threshold value means the exact opposite. Since there is no strong correlation between the size and re-reference interval of the cacheline, as observed in Figure 3, the threshold value that yields the best performance could only be empirically identified through sensitivity analysis. More specifically, the threshold value would be assumed to be static, and different threshold values would be explored. However, this assumption of a static threshold value would not maximize the performance in real systems, because the optimal threshold value varies with *not only* the application, but also the execution timeline of each application.

2.3.2.2 The Dynamically Adjustable Threshold Scheme

Figure 6 shows the *effective capacity* fluctuation and *physical memory usage* of streamcluster using the DRRIP replacement policy. The y-axis represents the cache size of the compressed LLC, which is physically 2MB, but can increase up to 8MB (logically), by overlaying a logical 16-way configuration onto a physically 4-way setup. The x-axis shows the timeline (cycles) for 0.9 billion cycles. In a single set of a compressed cache, the number of valid logical tags represents the *effective capacity*, while the total number of valid data segments in the data area represents the *physical memory usage*. Figure 6 highlights the fact that there are several sections of high effective capacity (indicating many small-size cachelines are being stored), and several sections of low effective capacity (indicating big-size cachelines are being stored). Obviously, the threshold value should be different for these sections.

Hence, we devise a Dynamically Adjustable Threshold Scheme (DATS), which dynamically changes the threshold value by simultaneously considering real-time *effective capacity* information and *physical memory usage* in a set. The threshold value is updated every time a new cacheline is inserted in a set and the threshold is calculated on a per-set

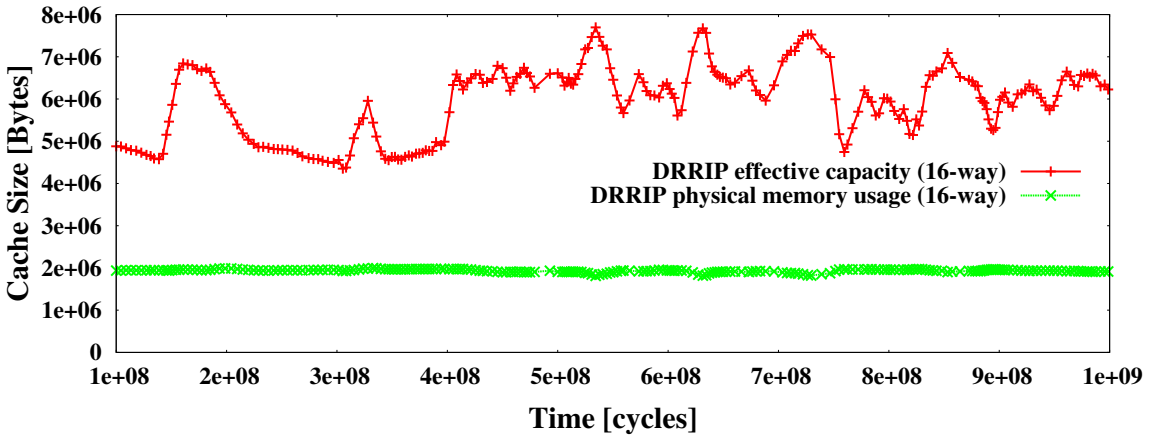


Figure 6: The effective capacity fluctuation and physical memory usage of streamcluster, when using the DRRIP replacement policy.

basis. In order to efficiently determine the dynamically changing T_h, while at the same time minimizing the implementation overhead within each set, we derive the Equation 1 as following:

$$T_h = \left\lceil \left(\left(\frac{NS_{uc} \times W_P}{W_L} \right) + NS_{uc} - \left[NS_{uc} \times \left(1 - \frac{NT_v}{W_L} \right) \right] \right) \times \left(\frac{Size_{total}}{NS_{uc} \times W_P} \right) \right\rceil \quad (1)$$

NS_{uc} indicates the number of segments occupied by an uncompressed cacheline, while W_P and W_L indicate the physical and the logical number of ways in a set, respectively. These three terms are constant, i.e, NS_{uc}=16, W_P=4, and W_L=16 in this study, and they are fixed at design time. NT_v indicates the number of valid tags within a set (i.e., the *effective capacity*) and $Size_{total}$ indicates the total number of valid segments in the data area of a set (i.e., the *physical memory usage*). Only these two variables are re-calculated when a new cacheline is inserted. The term ($NS_{uc} \times W_P$) represents the total number of segments in the data area (i.e., 64). The right-most term in the equation above ($Size_{total}$ / ($NS_{uc} \times W_P$)) represents the physical memory usage ratio. The left-most term (($NS_{uc} \times W_P$) / W_L) indicates the average number of segments in a logical way (i.e., 4), and this term is inserted as a bias. Based on the sum of this bias and NS_{uc}, the threshold T_h will decrease as NT_v increases and $Size_{total}$ decreases.

If NT_v is equal to W_L, it means all the tags are valid. In this case, the average size of cachelines in a set is relatively small, and we need to change the threshold value with a sufficiently small-size value. Furthermore, we also need to take into account the physical memory usage, in order to change the threshold more precisely, because, even if the tags are fully used in a set, the physical memory usage can range from 32 segments (128 B) to 64 segments (256 B). Therefore, we need to change the threshold with not only NT_v, but also $Size_{total}$. For example, when $Size_{total} = 32$ segments, then the threshold value will be $(4+16-16)\times(32/64) = 2$, while for 64 segments the threshold will be $(4+0)\times(64/64) = 4$, as per the equation above.

On the contrary, if NT_v is small (low effective capacity) – which indicates the presence of several big-size cachelines – we need to change the threshold with a larger-size value. In this case, we also need to take into account the physical data area size, because the small number of valid tags does not reflect the fact that there are big-size cachelines in the data area all the time. For example, if there are 8 valid tags, then the physical data area size can range from 16 segments (64 B) to 64 segments (256 B). Therefore, based on NT_v and $Size_{total}$, the threshold value will be $(4+16-8)\times(16/64) = 3$ for 16 segments, or $(4+16-8)\times(64/64) = 12$ for 64 segments. If there are only uncompressed lines (i.e, NT_v=W_P=4 and $Size_{total}$=64 segments), the threshold value will be $(4+16-4)\times(64/64) = 16$. Thus, the size-aware insertion mechanism will be disabled, because no cacheline occupies more than 16 segments.

Since the NT_v and $Size_{total}$ parameters (1) are defined at run-time using the tag area information, (2) they have to be read, in order to check for a hit or miss when a new request arrives, and (3) they do not require any additional storage elements (the threshold is updated whenever a cacheline is inserted), this technique can be implemented in hardware fairly easily and efficiently. Although the above equation is derived from heuristics, we will demonstrate that it effectively balances the classification rate of the various compressed cachelines.

In summary, the SAI policy classifies incoming cachelines as big-size or small-size, based on a size threshold, which varies dynamically during execution. The variable threshold value captures the run-time salient characteristics of each application. Upon size-aware classification, the SAI scheme assigns a corresponding RRPV value for the particular cacheline.

2.3.3 The Size-Aware hit-Promotion (SAP) Policy

As described in the previous sub-section, the SAI policy considers the cacheline's size information only once, at insertion time. However, our experiments have revealed that this initial consideration is not enough to maximize the effective capacity of the compressed cache. In the baseline RRIP framework [1], once a cacheline is inserted into the RRIP chain, the RRIP mechanism initiates promotions and/or demotions, depending on the hit/miss behavior of the line. This is the method by which RRIP manages the cacheline's locality information efficiently. In the ECM case, both the locality and the size information must be accounted for during this promotion/demotion process. Consequently, we devise a Size-Aware hit-Promotion (SAP) policy to tackle this balancing act.

Similar to SAI, the goal of the SAP policy is to enable a higher probability of eviction of big-size cachelines, while minimizing the potential conflict with locality-aware cache management. The SAP policy in ECM manages the promotion process, whereby a cacheline traverses the RRIP chain. In the conventional RRIP framework [1], when a cacheline is re-referenced its RRPV is promoted to zero (*near-immediate* re-reference interval prediction). On the other hand, the proposed SAP scheme adjusts the promotion step of the re-referenced cacheline (i.e., the change in RRPV), based on the line's classification as big-size or small-size (recall that the classification is performed using Equation (1), as defined in Section 2.3.2). Figure 7 depicts the operational principle of the proposed SAP policy. After a cacheline hit, SAP first checks the size information of the hit cacheline. If the line is classified as small-size, its promotion point in the RRIP chain is the same as in the traditional RRIP framework, i.e., the cacheline's RRPV is changed to zero. However, if the line

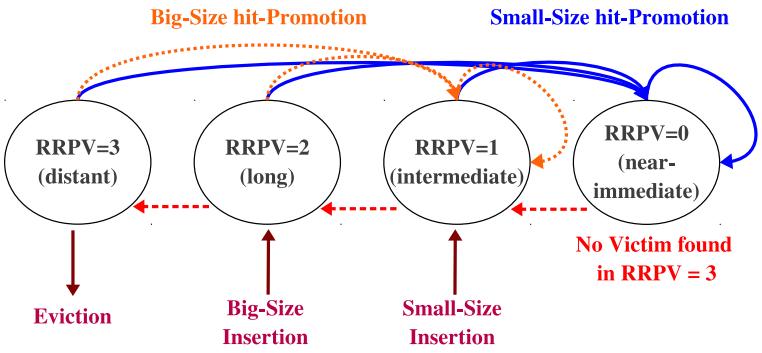


Figure 7: The ECM meta-data chain showing both the SAP policy transitions (top of diagram) and the SAI policy insertion points (bottom of figure), assuming a 2-bit (M=2) RRPV. The size information is considered both during insertion of the line in the cache, and after each hit (promotion process).

is classified as big-size, SAP changes the line's RRPV to a value higher than the promotion point of small-size cacheline, but no higher than the initial insertion point of big-size cachelines ($2^M - 2$). Specifically, as shown in Figure 7, the promotion point of big-size cachelines is set to an RRPV of one (*immediate* re-reference interval prediction). Thus, if a big-size cacheline happens to have the same locality information as a small-size cacheline, the SAP process ensures that the probability of eviction of the big-size cacheline is always higher than that of the small-size cacheline.

A similar philosophy may be applied to the demotion process. Under conventional RRIP, if a cache miss occurs and no cacheline is found with RRPV of $2^M - 1$, the RRPV of all cachelines is increased by one. However, unlike the promotion process, the demotion process involves size comparisons of all the cachelines in a set with a pre-defined threshold. This implies that the determination of the demotion point while accounting for the size of the cacheline involves a non-negligible implementation overhead. Additionally, increasing the RRPV of big-size cachelines to more than 1 has the effect of putting undue weight on the size information. In fact, in some cases we observed that adjusting the demotion point of big-size cachelines may even lead to a decrease in performance, because some big-size cachelines are evicted too quickly, before they get a chance to learn locality information.

Therefore, we apply the size-aware cache management concept only to the promotion process. For the demotion of a cacheline (upon a miss), a more sophisticated technique will be introduced later on, which considers size information in the eviction step without diluting the locality information.

Note that the SAP scheme only changes the promotion point in the RRIP chain, as compared to the size-agnostic conventional RRIP implementation. As a result, no significant alteration of the conventional framework is necessary to implement the SAP policy.

2.3.4 The Size-Aware Replacement (SAR) Policy

Figure 8 shows the RRPV values of the cachelines of one set in an 8-way set-associative cache. The letters 'a' to 'h' correspond to the eight ways of the set (i.e., each rectangle represents one cacheline of the set). The cachelines are ordered based on their RRPV values. The two right-most lines with RRPV of zero are at the "RRIP head," whereas the three left-most lines have RRPV of three, and are, therefore, at the "RRIP tail." In the baseline RRIP framework [1], the victim is selected among the cachelines whose RRPV is 2^M-1 (*distant* re-reference interval). These lines constitute what is referred to as the *eviction pool* (the three left-most lines in Figure 8). Conventional methods tend to select the left-most victim (as shown in the figure), or a random victim, if there is more than one cacheline in the eviction pool. This is a reasonable choice, because all the cachelines in the eviction pool have already been "studied" in terms of their re-reference interval, and they have been predicted to be re-used in the distant future. This reasoning, however, only captures the locality information.

Similar to the insertion (SAI) and promotion (SAP) policies, the ECM mechanism must account for both the size and locality information during the victim selection process as well, i.e., the size information must be utilized by the eviction policy. As previously explained, the RRPV of the cachelines in the eviction pool indicates distant re-reference intervals. Thus, we may consider the size information alone without affecting the locality. In other words, selecting the biggest-size cacheline from the eviction pool as the victim does

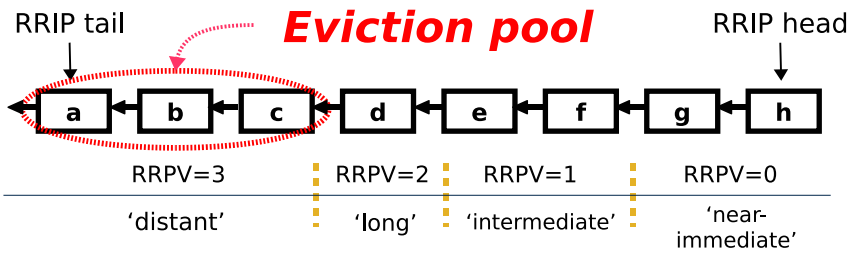


Figure 8: The RRPV values of the cachelines of one set in an 8-way set-associative cache. Each rectangle ('a' to 'h') corresponds to one way – i.e., one cacheline – of the set. The cachelines are ordered based on their RRPV values. A 2-bit (M=2) RRPV is assumed here.

not conflict with the locality information. This is precisely the premise of the proposed Size-Aware Replacement (SAR) policy, the operation of which is described in pseudo-code in Algorithm 1. Even though this is a very straightforward methodology, it is very effective in compensating for the weakness of the coarse-grained (binary) classification of the size information performed under the DATS policy. For example, let us assume that there are two cachelines having the same re-reference interval; one is quite bigger than the other, but due to the coarse-grained classification, they were classified under the same type of cacheline (either big-size, or small-size). This means that their probability of being selected as victims is also the same, even though their actual sizes are not the same. Under the proposed scheme, the bigger-size cacheline will be evicted earlier than the smaller one, if both cachelines find themselves in the eviction pool. In general, since the size information is applied only to the eviction pool, potential conflicts with the locality information are avoided.

The process of evicting the largest-size cacheline first has a two-fold advantage: (1) future cachelines have as much space as possible in the cache, and (2) the eviction penalty is reduced. The latter is a consequence of the fact that SAR minimizes the number of evicted cachelines. Without SAR, more than two cachelines would have to be evicted from the cache to the main memory if the size of the incoming cacheline happens to be larger than the victim cacheline (as described in Section 2.2.1). This, in turn, implies that the write-back buffer may stall more frequently. Instead, the SAR policy minimizes the

number of evicted cachelines by simply selecting the biggest cacheline among the victim candidates in the eviction pool.

The SAR mechanism selects the biggest-sized cacheline as a victim from the eviction pool, by using the 4-bit cacheline size information tag already present in the tag area. The SAR scheme requires 15 4-bit comparators, in order to identify the biggest-sized cacheline. The same comparators are also used by the DATS mechanism, in order to classify cachelines as big-size, or small-size. This is, in fact, the only overhead incurred. Since the proposed ECM mechanism is intended to be used on top of the RRIP framework [1] and a decoupled variable-segment cache architecture [5], all pertinent structures are already in place by said mechanisms. Other than the 15 comparators, no significant alteration or additional storage is required to implement the SAR policy.

2.3.5 The Size-Aware Eviction Scheduling (SAES) Policy

In most cases, the proposed SAR policy is very effective at minimizing the miss penalty. However, if the size of the new incoming cacheline happens to be larger than the combined size of all the cachelines in the eviction pool (i.e., larger than the sum of the cachelines' sizes), the SAR policy is not so effective. Not enough room is created for the new cacheline awaiting to be inserted, even after all the cachelines in the eviction pool are evicted. For example, in Figure 8, if the sum of the sizes of the cachelines in the eviction pool – 'a,' 'b,' and 'c' – is smaller than the size of the incoming cacheline, then cacheline 'd' should also be evicted after the demotion process. In order to tackle these problematic cases, we propose a modified replacement policy, called Size-Aware Eviction Scheduling (SAES). The main objective of the SAES policy is to minimize the miss penalty upon evictions, by considering the combined size of the entire eviction pool.

Using the same example of Figure 8, if the size of cacheline 'd' is equal to, or greater than, the size of the incoming cacheline, then evicting cacheline 'd' (which is not part of the current eviction pool) may be a better option than evicting all the cachelines in the eviction pool – 'a,' 'b,' and 'c' – and then 'd.' Based on this observation, the SAES policy extends

Algorithm 1 The operation of the SAR policy, assuming an M-bit RRPV.

```
/% finding a victim %/
while new cacheline size > vacant data segment size do
    if there is a cacheline whose RRPV = M-1 then
        find the biggest-size cacheline whose RRPV = M-1
        victim ← biggest-size cacheline
        vacant data segment size ← vacant data segment size + biggest-size cacheline size
    else
        for all the cachelines in the set do
            if RRPV ≠ M-1 then
                RRPV ← RRPV + 1
            end if
        end for
    end if
end while
```

Algorithm 2 The operation of the SAES policy, assuming an M-bit RRPV.

```
/% forming a new, size-guaranteed eviction pool %/
while new cacheline size > eviction pool size do
    for all the cachelines in the set do
        if RRPV ≠ M-1 then
            RRPV ← RRPV + 1
        end if
    end for
end while

/% finding a victim from the size-guaranteed eviction pool %/
while new cacheline size > vacant data segment size do
    /% reduced SAR policy %/
    find the biggest-size cacheline whose RRPV = M-1
    victim ← biggest-size cacheline
    vacant data segment size ← vacant data segment size + biggest-size cacheline size
end while
```

the range of the eviction pool by performing pre-demote operations until the sum of the cacheline sizes of this extended eviction pool is larger than the size of the new cacheline waiting to be inserted. The expanded eviction pool is called a *size-guaranteed eviction pool*. The detailed operation of the SAES policy (generating a size-guaranteed eviction pool and selecting a victim) is described in Algorithm 2. The SAES policy first compares the combined size of the cachelines in the eviction pool to the new cacheline's size. If the size of the new cacheline is smaller than the combined size of the cachelines in the eviction pool, then the regular SAR policy is used. In the opposite case, the SAES policy is used to create a size-guaranteed eviction pool by increasing the RRPVs of all the cachelines

in the set, until the combined size of the cachelines in the eviction pool is larger than the size of the new cacheline. A victim is then selected from the size-guaranteed eviction pool using a "reduced" SAR policy (second part of Algorithm 2). The SAES policy effectively minimizes the miss penalty on a cache miss by reducing the number of evicted cachelines.

2.4 Evaluation and Analysis

2.4.1 Experimental Methodology

2.4.1.1 Simulation Framework

A trace-driven simulator has been developed to evaluate the proposed ECM mechanism. Since the memory access sequence and the compression ratio are the most important factors for this evaluation, all traces have been extracted from the Simics full-system simulator [40], extended with GEMS [41], while simulating a quad-core processor with a two-level cache hierarchy. Ten multi-threaded benchmarks from the PARSEC benchmark suite [2] are selected, and each benchmark runs for 300 million instructions. The L1 caches use an LRU replacement policy, and our study focuses on the cache management of the LLC. All the details of the simulation parameters are described in Table 1. In addition to performance evaluation, we also perform energy consumption simulations by integrating power consumption specifications into our trace-driven simulator. The energy parameters of the SRAM-based LLC and the DRAM-based main memory system are obtained from CACTI [42] and a commercial DRAM data sheet [43], respectively.

2.4.1.2 The Compression Technique Employed in the LLC

This chapter assumes that the targeted baseline architecture uses compression only in the LLC. No compression is assumed in the L1 caches, nor in main memory. As a compression algorithm for the LLC, we choose Frequent Pattern Compression (FPC) [7] – a bit-level compression algorithm – because its compression performance is relatively high, with reasonable compression overhead, both in terms of delay and implementation cost. The energy parameters of the FPC compression algorithm were obtained from [16]. We apply FPC within a word, so a compressed cacheline's size varies from 8 B (maximally

Table 1: Simulated system parameters for evaluating ECM.

Number of CMP cores	4
Processor core type	UltraSPARC-III+, 2 GHz
L1 caches (private)	I- and D-caches 32 KB, 4-way, 64 B
L1 response latency	3 cycles
L2 caches (shared)	2 MB, 4-way, 64 B, NUCA, MESI
L2 response latency	20 cycles
L2 read hit overhead	5 cycles for decompression
L2 writeback buffer	8 entries
Compaction overhead	16 cycles
DRAM memory	DDR2 4 GB
Main memory response latency	450 cycles

compressed) to 64 B (uncompressed). The maximally compressed cacheline occupies only 2 segments, while the uncompressed cacheline occupies 16 segments in the decoupled variable-segment LLC, as illustrated in Figure 1. A 2 MB physically 4-way LLC is used. We set the maximum number of ways in a set to 16 (logical ways), so the effective capacity can increase up to 8 MB, which is a significant improvement. Without loss of generality, these parameters help demonstrate the efficacy of the proposed ECM mechanism.

A critical operation in compressed LLCs is data compaction, which is required when the cache has room for upcoming requests (read misses, write hits, and write misses), but not in consecutive segments. In the case of compaction on a read miss, we do not account for the compaction overhead, because the compaction can be done while the data is being fetched from the main memory. This implies that no additional delay is incurred due to compaction. However, for both write hits and misses, the cacheline size changes and compaction is necessary in most requests. Without completing a compaction, data cannot be written in the cache. Thus, our evaluation will accurately account for the compaction overhead (shown in Table 1) of write requests.

2.4.2 Workload Characteristics

We begin our evaluation by first analyzing the salient characteristics and cache-related behavior of the application workloads used in the study of the proposed ECM mechanism.

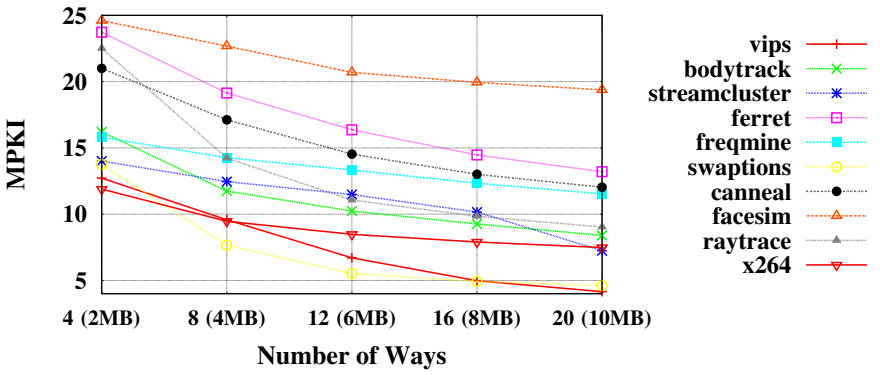
These key characteristics are the set-associativity sensitivity, the achievable compression ratios, the distributions of cacheline sizes, and the locality attributes of each cacheline size. When we overlay a logically 16-way cache on a physically 4-way 2 MB LLC using compression, the maximum effective capacity is increased up to 8 MB. This effect is, essentially, the same as increasing the number of ways without changing the number of sets, as shown in Figure 9(a). The y-axis shows the number of misses per thousand instructions (MPKI), while the x-axis shows the number of ways and the physical cache size. This set-associativity sensitivity directly shows how much we can reduce the number of misses by increasing the effective capacity of each application. Even though setting the number of ways to 20 enhances performance over a 16-way setup, we set the number of logical ways to 16 in all the evaluations, because this design choice strikes the most efficient tradeoff between obtained performance and implementation overhead.

Figure 9(b) shows the achieved compression ratios and the distributions of the compressed cacheline sizes for the 10 evaluated benchmarks. A lower compression ratio indicates higher compressibility. The benchmark `vips` – which is the most compressible application – has over 50% of the total cachelines compressed within 2 segments (8 B). As previous compression studies have indicated, the main reason for this high compressibility is the large number of zero values. The 16-segment bars (64 B) indicate the incompressible cache line ratio; it accounts for 19% of `vips`. On the other hand, the benchmark `x264` shows the lowest compressibility. Only 20% of the total cachelines are compressible to within 2 segments, while almost 40% of the cachelines are incompressible.

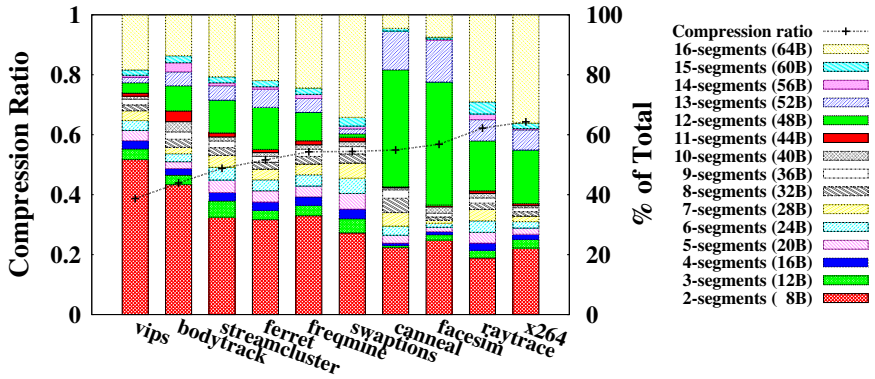
2.4.3 Assessing the Size-Aware Cache Management Policies

2.4.3.1 The Effects of Compression and Size-Awareness

The proposed ECM mechanism is first compared to LRU without compression (LRU-u), LRU with compression (LRU-c), DRRIP [1] without compression (DRRIP-u), and DRRIP with compression (DRRIP-c), in terms of the effective capacity, cache miss-count reduction, and system performance. For ECM and DRRIP, 3-bit RRPVs are used. Based on



(a) Miss behavior sensitivity to physical cache set-associativity

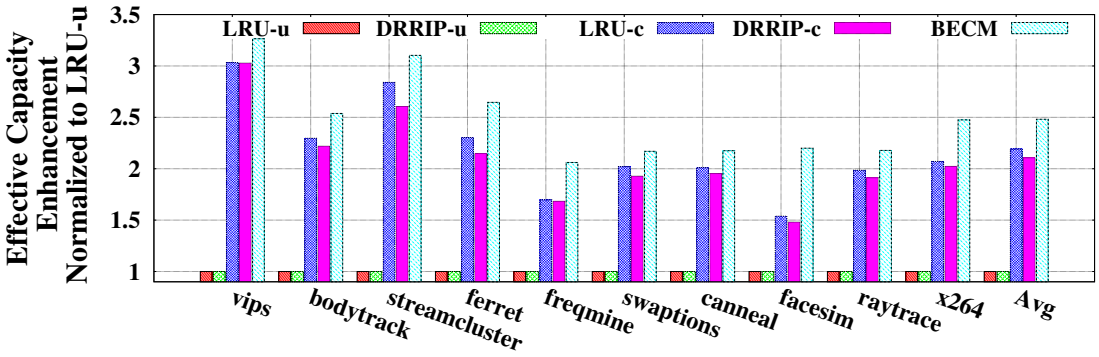


(b) Achievable compression ratios and distributions of line sizes

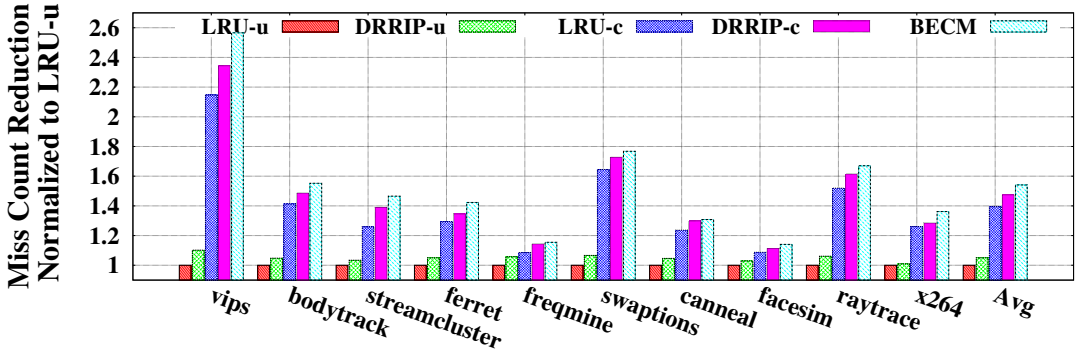
Figure 9: The salient cache-related characteristics and behavior of the application workloads used in this study.

extensive experimentation and sensitivity analyses, we set the big-size insertion point to an RRPV of $2^3 - 2$ and the small-size insertion point to $2^3 - 3$, because these values maximize the average percent miss count reduction. In order to isolate the contributions of the SAP and SAES policies, we first examine a **"Baseline ECM" (BECM)** scheme, which includes only two of the four main policies presented in this work; namely, SAI and SAR (but not SAP, or SAES). This BECM setup was initially presented in [4].

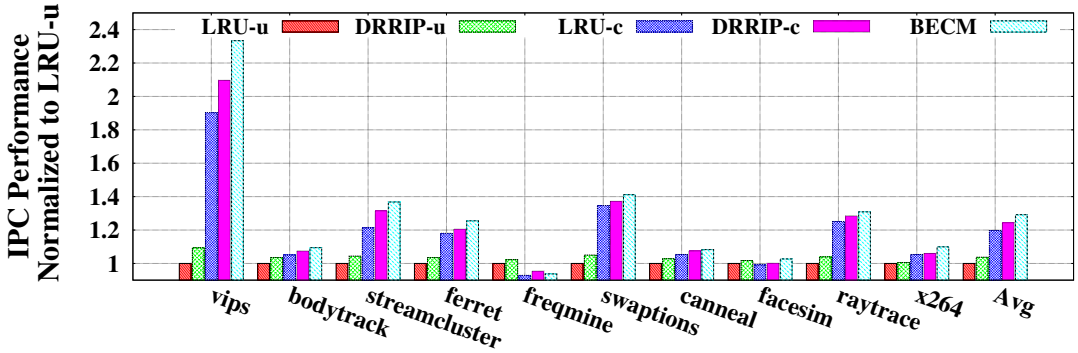
Figure 10(a) shows the effective cache capacity – normalized to LRU-u – assuming a 2 MB uncompressed LLC. Merely applying data compression without changing the replacement policy improves the effective capacity between 1.5 to 3 times, depending on the application. The BECM technique improves the effective capacity by an average of 2.5



(a) Effective cache capacity improvement



(b) Cache miss-count reduction



(c) Overall system performance

Figure 10: Assessing the overall performance of a "Baseline ECM" (BECM) mechanism, which only employs the SAI and SAR policies, but not SAP/SAES.

times, by simply considering the size information, while the LRU-c and the DRRIP-c techniques improve the capacity by an average of 2.2 and 2.1 times, respectively. The `vips` benchmark shows the highest capacity enhancement under all replacement policies, while `facesim` shows the least capacity enhancement under LRU-c and DRRIP-c. This indicates that the effective capacity enhancement exhibited by an application is not always proportional to the compression ratio, because some big-size cachelines may stay longer in the cache (if they have higher locality) than small-size cachelines.

Figure 10(b) shows results pertaining to the cache miss count reduction, normalized to LRU-u. The `vips, swaption,` and `raytrace` benchmarks are ranked first, second, and third, with 156.6%, 76.8%, and 67.1% miss count reductions, respectively – as compared to LRU-u – under the BECM mechanism. These are the applications that are most sensitive to the set-associativity and physical cache size, as indicated in Figure 9(a). This attribute implies that their benefits mostly emanate from enhancements in the effective cache capacity. We also found that the cache miss reduction is not directly proportional to the enhancement of the effective capacity, because of the conflicts between the locality and the size information. For example, the `streamcluster` and ferret benchmarks show the second- and third-best improvements in effective capacity, respectively. However, their miss count reductions are lower than those of `swaption` and raytrace. On the average, BECM achieves a 54.1% miss count reduction, as compared to LRU-u.

Finally, we compare the overall system performance (in terms of Instructions Per Cycle, IPC) of each configuration, as shown in Figure 10(c). The y-axis shows the IPC normalized to LRU-u. The `vips` benchmark shows the best IPC performance improvement under all investigated techniques: 90.2% under LRU-c, 109.6% under DRRIP-c, and 133.3% under BECM. This is mainly due to the large miss count reduction under all techniques. We observe significant performance enhancement in most applications, except `facesim` and `freqmine`. We even observe some performance *degradation* in `freqmine`, regardless of the replacement policy. These problematic benchmarks extract very little benefit from

compression, so they suffer from excessive compression/decompression overhead. For such applications, an adaptive cache compression technique [5] would be a useful solution for minimizing the performance degradation. On the average, the BECM mechanism enhances the system performance by 29.2%, 7.9%, and 3.9% over LRU-u, LRU-c and DRRIP-c, respectively.

2.4.3.2 The Effect of Adding the SAP Policy to SAI+SAR

The proposed ECM architecture consists of a combination of individual techniques: SAI, SAP, and SAR+SAES. In the previous sub-section, we analyzed the performance of BECM, which is a reduced ECM design employing only SAI and SAR [4]. The next few sub-sections will focus on further enhancing ECM's performance by incrementally applying the SAP and SAES policies. To distinguish it from BECM, the full-fledged ECM design with all presented policies will be henceforth referred to as the **"Enhanced ECM" (EECM)**. Before proceeding to assess EECM, we first analyze the effect of adding the SAP policy alone (without SAES) to BECM (i.e., adding SAP to SAI+SAR).

The SAP policy can be orthogonally applied to BECM to further enhance the performance of the system, but the proper promotion points should be decided after appropriate sensitivity analysis. Toward this end, we integrate the SAP policy into BECM, and statically change the promotion point of big-size cachelines to investigate the impact on

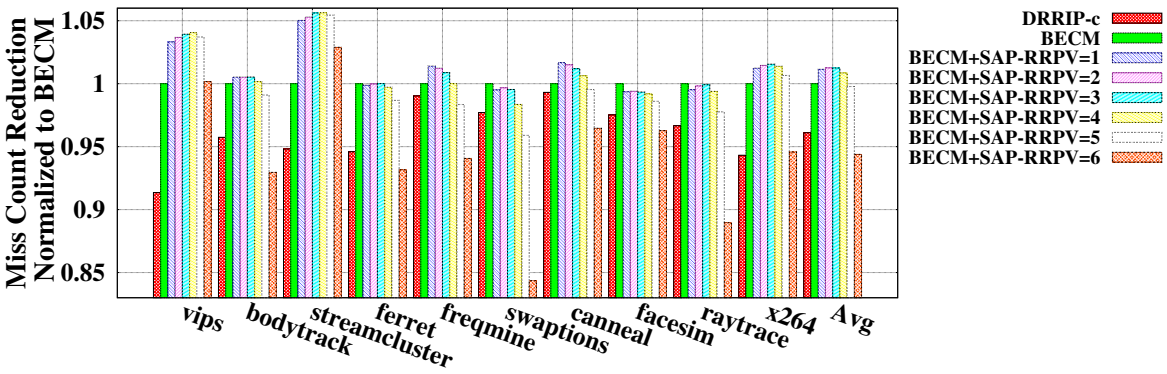


Figure 11: Investigation of the effect of adding the SAP policy to BECM (i.e., adding SAP to SAI+SAR). The RRPV promotion point of big-size cachelines is varied from 1 to 6, in order to identify the most effective promotion point under SAP. A 3-bit RRPV is assumed.

miss-count reduction. Note that the promotion point of small-size cachelines is fixed to RRPV=0 to ensure optimal *locality* awareness. By then varying the promotion point of big-size cachelines, one may assess the effect of *size* awareness and identify the optimal balance between locality and size awareness. Since the assumed RRPV bit-width (i.e., the width of the M-bit register) is 3, the RRPV takes values from 0 to 7, where 0 indicates near-immediate re-reference interval and 7 indicates distant re-reference interval. In our experiments, the RRPV promotion point of big-size cachelines is varied from 1 to 6, in order to give big-size cachelines an increasingly larger probability of eviction. Figure 11 shows the simulation results, normalized to BECM. One may observe that the `ferret`, `swaption`, `facesim`, and `raytrace` applications do not show appreciable miss-count reductions, as compared to BECM. This result indicates that over-consideration of size information may pollute locality information and worsen the cache performance. Moreover, some performance degradation is also observed in several applications, if we set the promotion point (RRPV in RRIP chain) of big-size cachelines closer to 2^M-1 (*distant* re-reference interval), because of a very short travel time within the RRIP chain. However, the `vips`, `streamcluster`, and `x264` applications show the higher miss-count reductions when we set the promotion point of the big-size cachelines to RRPVs of 3 or 4. On the contrary, the `freqmine` and `canneal` applications show the higher miss-count reductions under RRPV insertion points of 1 or 2. Even though the optimal promotion points vary with the applications, setting the RRPV promotion point to 2 shows the best average miss count reduction (1.3%), as compared to BECM. For the rest of the chapter, unless otherwise stated, we only provide results with the promotion point of big-size cachelines set to RRPV=2.

2.4.3.3 Performance Analysis of the Enhanced ECM Mechanism (SAI+SAR+SAP+SAES)

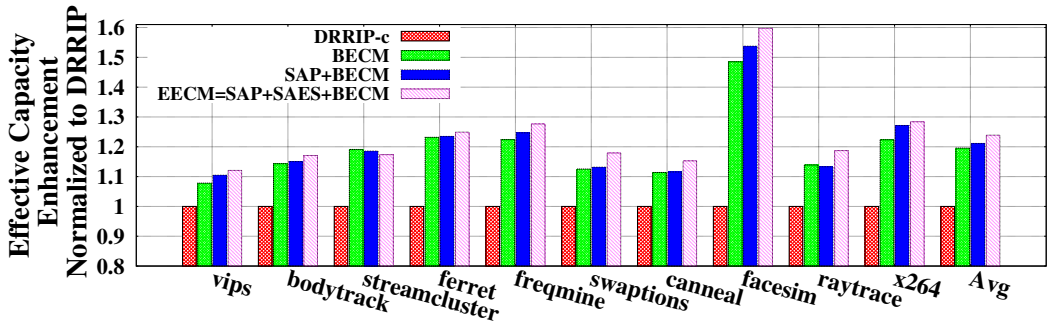
This sub-section will assess the combined effects of all ECM techniques. The baseline reference architecture is DRRIP with compression (DRRIP-c), which is the state-of-the-art size-agnostic cache management policy. In order to appreciate the effects of each individual policy of the ECM architecture, we also include the BECM design (SAI+SAR) [4],

SAP+BECM (as described in the previous sub-section), and the complete, Enhanced ECM (EECM), which corresponds to SAP+SAES+BECM.

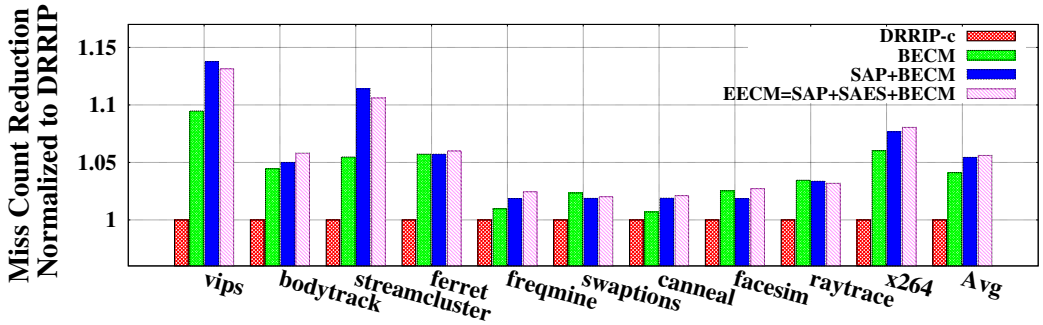
Figure 12 shows the performance comparison results, normalized to DRRIP-c. In addition to the effective cache capacity enhancement, the miss-count reduction, and the IPC improvement, we also analyze the write-back delay reduction, which is the main benefit of the SAES policy. As expected, the SAP+BECM and the EECM (SAP+SAES+BECM) designs improve on almost all evaluation metrics, as compared to DRRIP-c and BECM.

The *vips* and *streamcluster* benchmarks show relatively higher miss-count reductions with SAP, although their effective cache capacity improvements are not significant. This results in an overall system performance improvement of 16.3% and 8.2%, respectively, because SAP helps to maintain high-locality big-size cachelines longer in the cache than low-locality big-size cachelines. In other words, SAP accelerates the eviction of low-locality big-size cachelines, by reducing the travel time of the big-size cachelines in the RRIP chain. The *swaption and facesim* applications do not improve the cache miss-count and system performance when compared to the BECM, although they exhibit effective capacity improvements. As previously mentioned in Section 2.4.3.2, the over-consideration of size information may pollute the locality information and, thus, worsen the cache performance in these applications.

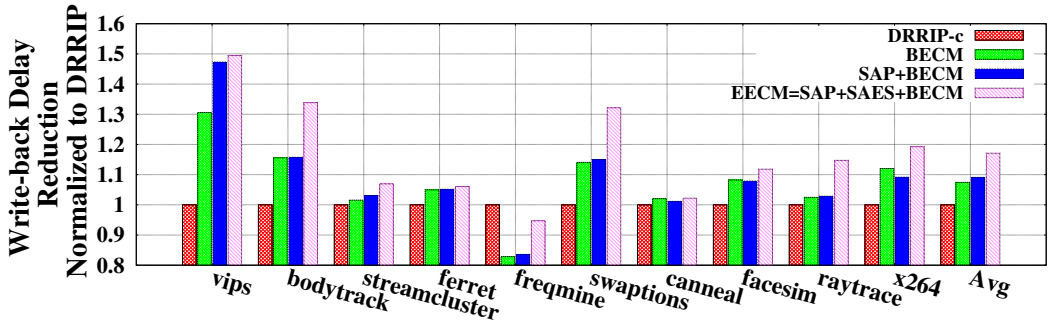
The SAP policy has very little effect on the write-back delay performance, except with `vips`, because the write-back delay is primarily proportional to the number of evicted cachelines, which SAP does not significantly affect. The write-back delay of `vips` is reduced mainly due to the large reduction in cache misses. Note that `freqmine` is the one application where the compression/decompression overhead exceeds the benefits of BECM, as shown in Figure 10. Although the performance of SAP+BECM is worse than DRRIP-c, the SAP policy enhances the performance, compared to BECM. In summary, by giving a little more weight to the size information than to the locality information in SAP, the SAP+BECM setup enhances the average performance of the system by 4.1%, compared



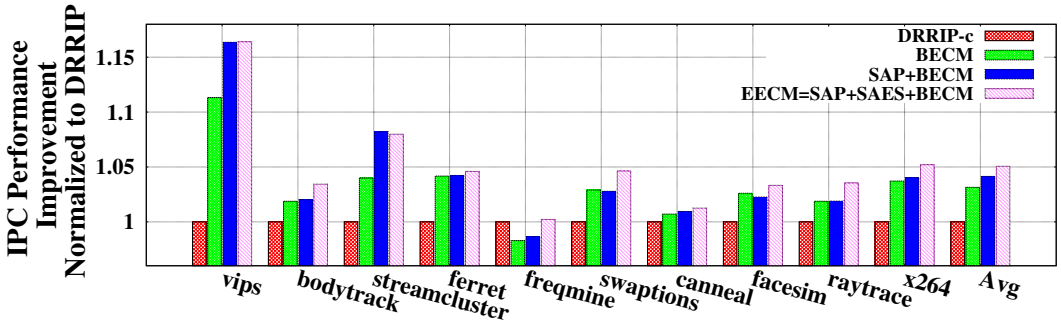
(a) Effective cache capacity improvement



(b) Cache miss-count reduction



(c) Write-back delay reduction



(d) Overall system performance

Figure 12: Performance evaluation of the "Enhanced ECM" Mechanism. In order to isolate the benefits of each individual policy, the performance results of designs with incrementally fewer policies are also presented. All results are normalized to DRRIP-c.

with DRRIP-c, and 0.96%, compared with BECM. Although these average improvements are not very significant, the addition of SAP is still meaningful, because its implementation cost is almost negligible.

The SAES policy is also helpful in further enhancing the effective cache capacity, as shown in Figure 12, even though improvements to the cache miss-count are not noticeable. For example, the `facesim` benchmark shows the highest effective capacity enhancement, but its cache miss-count is only slightly reduced, much like other applications. In fact, the `vips, streamcluster,` and `raytrace` applications exhibit a slight increase in the cache miss-count. This is attributed to the unbalanced consideration between size and locality in these applications.

Unlike SAP, the main focus of SAES is the reduction of the eviction penalty by repeatedly applying demote operations before selecting a victim, based on size information. Figures 12(c) and (d) demonstrate that the SAES policy significantly reduces the eviction penalty by minimizing the number of evicted cachelines per cache miss. This translates into an improvement in overall system performance. Consequently, the EECM design – which combines all policies: SAI+SAR+SAP+SAES – shows the highest average performance improvement at 5.1%, as compared to DRRIP-c.

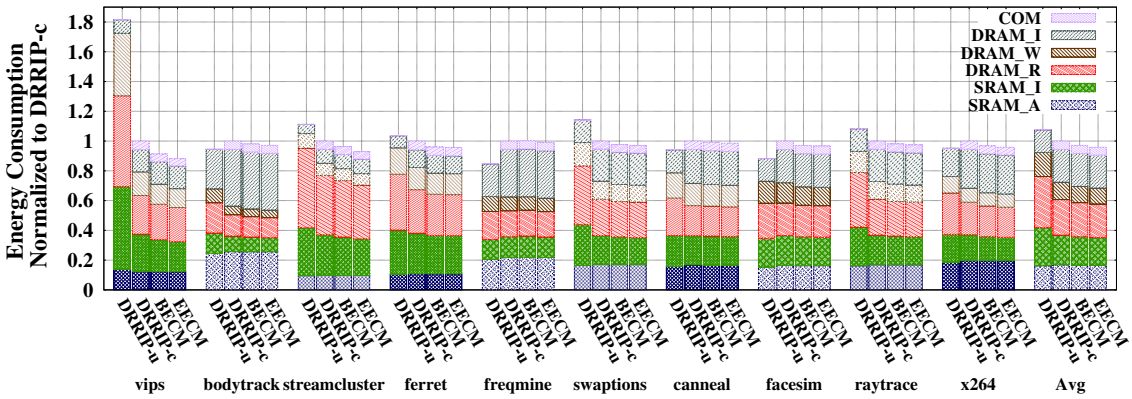


Figure 13: Analysis of the energy consumption of the proposed mechanisms. The six major components of the total energy consumption are described in Table 2. The results are normalized to the energy consumption of DRRIP-c.

Finally, the energy consumption of the designs under evaluation is compared in Figure 13. The results are normalized to the energy consumption of DRRIP-c. The total energy consumption is decomposed into six components that is comprising the total energy consumption of the various cache management schemes, as shown in Table 2. The compression/decompression operations represent about 6% of the total consumed energy in most applications. Overall, the energy reduction demonstrated by EECM is proportional to its performance improvement, because of the reduced number of main memory accesses and reduced execution times. The EECM setup reduces the energy consumption for all applications, except `freqmine, canneal,` and `facesim`. In these applications, the energy consumed during compression/decompression and compaction – including the additional energy due to the extra tag area – exceeds the small energy reduction in the memory components. However, in all other applications, the significant reduction in the DRAM and SRAM leakage energy consumption outweighs this energy overhead. On the average, DRRIP-c reduces the energy consumption by 6.9%, as compared to the DRRIP-u (a direct consequence of compression), while the proposed EECM architecture further reduces the energy consumption by 4.1%, as compared to DRRIP-c.

Table 2: The six major components of total energy consumption.

Terms	Description
COM	Compression / decompression energy
DRAM_I	DRAM leakage energy
DRAM_W	DRAM write energy
DRAM_R	DRAM read energy
SRAM_I	SRAM leakage energy
SRAM_A	SRAM active energy

To sum up, our proposed EECM increases the effective capacity and reduce the number of evictions while minimizing conflict with locality by considering cacheline size and locality information together. The Figure 14 shows the number of evicted cachelines of `facesim` (which shows most improvement in effective capacity) when a miss is occurred, using DRRIP-c, BECM, and EECM in a compressed LLC. The x-axis shows the timeline (cycles). Note that, in an uncompressed cache, the number of evicted cachelines is always

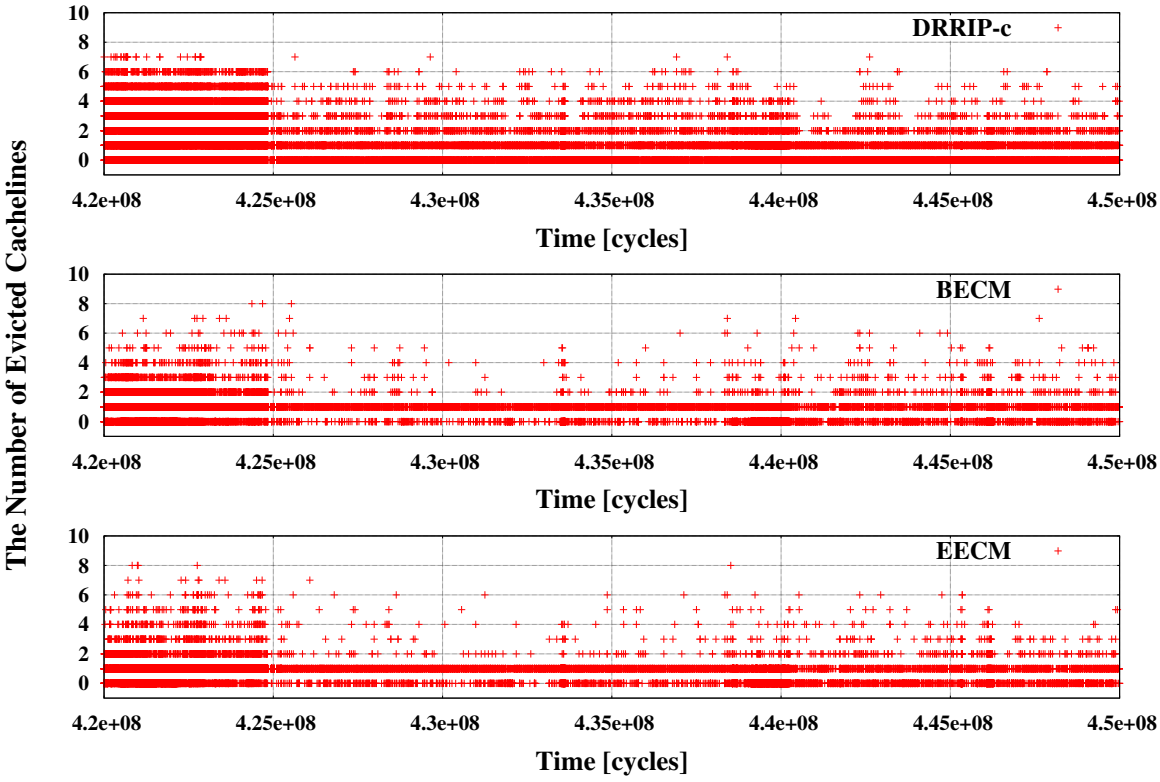


Figure 14: The number of evicted cachelines of `facesim`, when a miss is occurred.

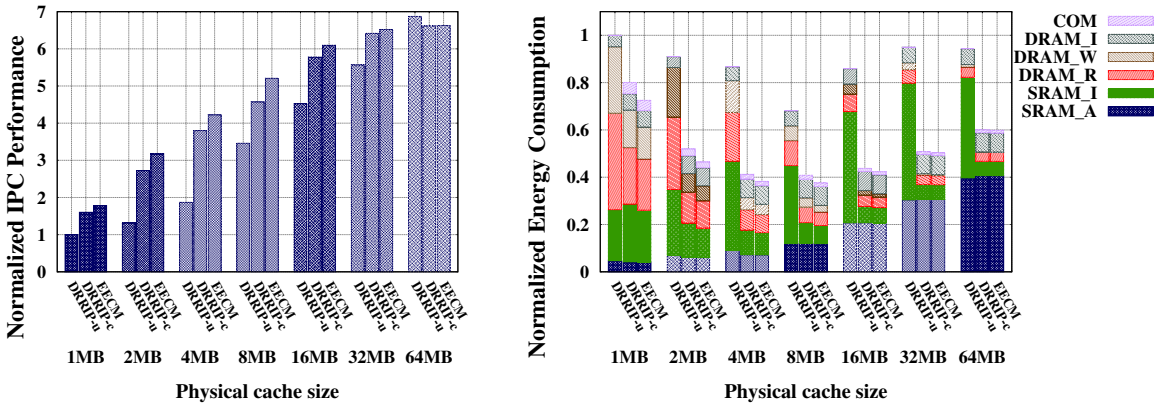
one when a miss is occurred. However, in a compressed cache, the number of evicted cachelines can be varied from zero (if there is enough room for new cacheline) to eight (to make enough room by evicting the most compressible cachelines that is compressed within 2 segments – 8 bytes). When we use a DRRIP-c, which does not consider the size information, there are a lot of zero evictions when a miss is occurred. In other words, there is already enough vacant data segments for a new cacheline. This implies that the physical memory (data segment) is inefficiently used, as compared to an uncompressed cache. Also, there are a lot of periods which show more than two cachelines evictions due to the lack of considering the size information. The number of these zero eviction and more than two cachelines evictions can be reduced by considering the size information, as we can see in the proposed BECM and EECM; the number of more than four evictions is reduced remarkably when a miss is occurred. These reduced the number of eviction allows EECM to outperform DRRIP-c by reducing the miss penalty.

2.4.4 Performance & Energy Consumption Sensitivity to Cache Size and Logical Set Associativity

Though the proposed ECM architecture successfully enhances the performance and energy consumption, it is important to explore other key design parameters, in addition to the replacement policy. Hence, we investigate the performance and energy consumption sensitivity of the full EECM setup (ECM with all policies presented in this chapter) on the physical cache size and the logical set associativity. Figure 15 compares the IPC performance and the energy consumption by varying the physical LLC size, from 1 MB to 64 MB. The `vips` benchmark, which shows the highest IPC improvement, is used in this evaluation. The effective capacity of each physical cache size can be increased up to four times, because all LLCs are logically 16-way overlaid on top of a physically 4-way cache. All results are normalized to DRRIP-u with 1 MB LLC.

As shown in Figure 15(a), the performance of all configurations improves as the physical size of the LLCs increases. The performance of DRRIP-u starts to saturate at 64 MB LLC (not shown in the graph – which stops at 64 MB – but verified by further experiments), indicating that cache sizes beyond 64 MB are not helpful in further reducing the number of cache misses. On the other hand, the performance of ECM starts to saturate at LLC sizes of around 16 to 32 MB, because its effective capacity has already reached 64 MB. Beyond this point (cache sizes larger than 32MB), the compressed LLCs show less performance improvement – or even slight performance degradation – because of the compression overhead. This implies that ECM works very well even when the physical cache capacity is not large enough to fit the data set of the application. Conversely, ECM allows for the reduction of the physical cache size to one half (or a quarter) of the original size required in uncompressed LLCs, without severe performance degradation.

From the perspective of the energy consumption – illustrated in Figure 15(b) – increasing the physical LLC size is accompanied by both positive and negative effects. Increasing the physical LLC size results in an increase in the energy consumption per LLC access,

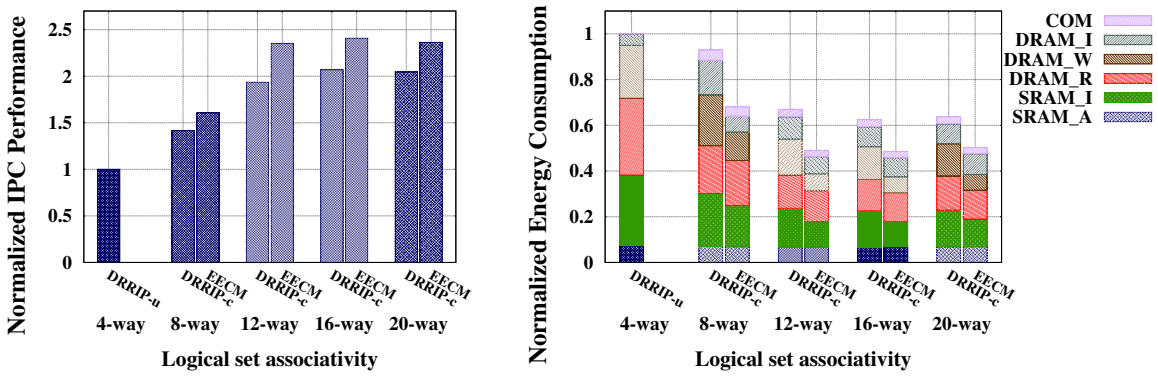


(a) Overall system performance (b) Energy consumption

Figure 15: The performance and energy consumption sensitivity of `vips` to the physical LLC size, assuming a logically 16-way LLC configuration for ECM and DRRIP-c. All the results are normalized to a physically and logically 4-way DRRIP-u setup with 1 MB uncompressed LLC.

but a reduction in the number of main memory accesses. Thus, by increasing the physical LLC size, the total energy consumption in the cache (SRAM_A in Figure 15(b)) increases, while the DRAM energy consumption (DRAM_R and DRAM_W) decreases. The reduced energy consumption in the DRAM_R and DRAM_W components outweighs the energy consumption increase in the cache, until the cache size is 8 MB. However, when the cache size is 16 MB and larger, the increased energy consumption in the cache dominates the energy reduction in other components. As a result, the entire energy consumption due to the memory system increases, as the physical LLC size exceeds 8 MB.

The sensitivity of ECM to the *logical* set associativity was also investigated. Specifically, five different logical set associativities were analyzed: 4-way, 8-way, 12-way, 16-way, and 20-way. Since a 2 MB physically 4-way LLC is used as a baseline, the effective capacity increases up to 4 MB, 6 MB, 8 MB, and 10 MB, respectively. Figure 16(a) shows the performance variations in each configuration. Increasing the logical ways definitely helps in enhancing the effective cache capacity, which, in turn, results in overall system performance improvements. However, the performance enhancement starts to saturate from the 16-way configuration, because the need to hold more than 16 ways in a cache set is rare



(a) Overall system performance

(b) Energy consumption

Figure 16: The performance and energy consumption sensitivity of `vips` to the logical set associativity, assuming a 2 MB LLC. All the results are normalized to a physically 4-way DRRIP-u setup with 2 MB uncompressed LLC.

in most applications, even in a compressed cache. Hence, the 16-way setup is the most cost-efficient configuration.

Finally, Figure 16(b) analyzes the energy consumption sensitivity of ECM to the logical set associativity. Increasing the number of logical ways requires more space in the cache to store the extra tags. This extra space slightly increases the energy consumption per cache access. As shown in the figure, the SRAM_A component increases slightly as the number of logical ways increases. However, this overhead is almost negligible compared to the energy consumption in the other components. For example, the energy consumption in the DRAM (DRAM_W and DRAM_R) is significantly reduced due to the decreasing numbers of DRAM accesses resulting from the increases in the effective cache size. Once again, the 16-way cache configuration used in this work is shown to be the most cost-efficient choice.

2.5 Conclusion

The "memory wall" phenomenon is one of the biggest obstacles faced by microprocessor designers. The growing gap between processor and memory performance elevates the criticality of the cache hierarchy and magnifies the importance of the Last-Level Cache

(LLC). While the LLC size continues to grow with each technology generation, application demands also increase. One approach to increasing the effective cache capacity without increasing the physical capacity is to compress the LLC.

The cacheline size in compressed caches is variable and depends on the achievable compression ratios. This work investigates the importance of cacheline size in the cache management policy. Currently, there are no replacement policies tailored to compressed LLCs, and all existing approaches are line-size-agnostic. The proposed Effective Capacity Maximizer (ECM) mechanism is a size-aware cache management policy specifically geared toward compressed LLCs. The ECM scheme aims to maximize the performance and minimize the energy consumption of the cache, through a quartet of policies working in tandem: (1) Size-Aware Insertion (SAI), (2) Size-Aware hit Promotion (SAP), (3) Size-Aware Eviction Scheduling (SAES), and (4) Size-Aware Replacement (SAR).

Extensive evaluation using memory traces extracted from real application workloads running on a full-system simulation environment demonstrate the efficacy and efficiency of ECM. Specifically, the proposed ECM architecture exhibits an average effective cache capacity increase of 23.9%, and an average cache miss reduction of 5.6%, as compared to the state-of-the-art DRRIP framework [1]. These enhancements translate to an average system performance improvement of 5.1% over DRRIP. Similarly, ECM is shown to lower the memory system's energy consumption by 4.1%, on average, compared to DRRIP.

CHAPTER 3

HOT-CACHELINE PREDICTION FOR DYNAMIC EARLY DECOMPRESSION IN COMPRESSED LAST-LEVEL CACHES

Since the advent of the Chip Multi-Processing (CMP) paradigm, more cores have had to share the already constrained memory resources, fighting for use of buses and memory to not only access data, but also to maintain cache coherence and even share data between the many cores. Such techniques, which have become a staple in todays architectures, serve to increase an already wide gap between processor and memory performance. In order to bridge this performance gap, a high-performing memory structure is needed. Current architects have been partially solving the problem by allocating more and more space to on-chip caches, but that has not been shown sufficient to overcome the increase in the use of memory resources due to increasing working-set sizes of applications and the additional concurrency overhead of multi-core programming. As a result, determining how to use the fixed space allocated to memory most effectively is one of the most important research challenges to overcome in the field of microprocessor design.

An attractive solution to overcoming this processor-memory performance gap is using compression in the memory hierarchy, as it is able to increase effective memory capacity while using the same amount of physical space. This increase in memory capacity allows for the physical memory to hold a larger working set and improve system performance significantly. Memory compression has been shown to be very promising and it has even been adopted commercially; the industry giant Apple Inc. has recently introduced compressed memory to its new operating system (OS X Mavericks), in order to improve responsiveness under load [44]. This shows that the industry is becoming more aware that using compression in memory is an attractive way to increase performance with low overhead. In addition to *software-managed* compression (as used in OS X Mavericks),

the concept of *hardware-enabled* compression has also garnered attention, within the context of on-chip caches. The potential increase in effective capacity has led researchers to develop various compressed Last-Level Cache (LLC) architectures by designing efficient compression algorithms [7, 8, 9, 10, 11, 12, 45, 46, 47, 48], compression-aware cache structures [13, 5, 8, 14, 15, 16, 17, 12, 49, 50], and compression-aware cache management policies [4]. However, blindly applying compression everywhere is not always a good idea, as compression has some overhead designers must take into consideration.

Data compression often helps memory-constrained applications, but, designed incorrectly, it can incur performance degradation. Using data compression techniques has some overhead, such as compression/decompression latency, compaction overhead, and address remapping [7, 8, 9, 10, 11, 12, 13, 5, 14, 15, 16, 17]. The fundamental challenge of this work is to avoid, or minimize, the overhead when using compression techniques. In particular, the decompression latency for read hits degrades system performance significantly, because the decompression latency increases the memory access latency. Although prior efforts have tried to avoid the read-hit-decompression overhead by using compression adaptively (e.g., in [14] and [5]), the overhead remains significant in those techniques.

To address the possibly debilitating issue of excessive read-hit decompression overhead in compressed LLCs, we hereby propose a novel mechanism, called **Hot-cacheline Prediction for Early decompression (HoPE)**. The main contributions of this work revolve around three techniques, which work synergistically and in unison within the HoPE framework:

- **Hot-cacheline Prediction (HP)** – HP dynamically identifies frequently used cachelines (termed "hot cachelines") with minimal hardware overhead, by extending existing cache replacement mechanisms.

- **Early Decompression (ED)** – ED decides which of the identified hot-cachelines should be eagerly decompressed, in order to minimize the adverse effect on effective

cache capacity, while – at the same time – increasing the read-hit performance.

- **Hit-history-Based Insertion (HBI)** – HBI assists the HP technique by filling the cache with cachelines that are predicted to exhibit higher locality.

HoPE is very lightweight in terms of hardware implementation, and all mechanisms involved mostly re-use existing components within the compressed LLC infrastructure.

To evaluate the effectiveness of the proposed HoPE mechanism, we run extensive simulations on memory traces obtained from multi-threaded benchmarks running on a full-system simulation framework. The simulations demonstrate that HoPE reduces the read-hit decompression penalty in compressed LLCs by an average of 64.1%, 64.4%, and 64.5%, when compared to compressed caches using the Least Recently Used (LRU) replacement policy, the Dynamic Re-Reference Interval Prediction (DRRIP) scheme [1], and the recently introduced Effective Capacity Maximizer (ECM) mechanism [4], respectively. This reduction in read-hit penalty increases system performance by an average of 9.8% over LRU, 6.2% over DRRIP, and 4.6% over ECM. The system performance improvement obtained by the proposed HoPE mechanism is primarily due to the reduction in the read-hit penalty.

The rest of the chapter is organized as follows: Section 3.1 provides background information in memory compression techniques and motivates the need for an eager (early) decompression mechanism in compressed LLCs. Section 3.2 describes related work in the domain of memory/cache compression. Section 3.3 delves into the description, implementation, and analysis of the proposed HoPE architecture. Section 3.4 describes the employed evaluation framework and presents the various experiments and accompanying analysis. Finally, Section 3.5 concludes the chapter.

3.1 Background and Motivation

3.1.1 Compression Techniques in Memory Systems

Compression techniques have been used to primarily reduce memory traffic and increase the effective memory capacity of all levels of the memory hierarchy. These levels includes caches, main memory, and secondary storage. Unlike normal data compression in the file system of secondary storage, data compression in the memory system has some limitations, such as compression/decompression overhead and address mapping, due to the variable compressed data size. Depending on the targeted level of the memory hierarchy, the design goals of data compression change.

At the cache level, access latency and effective capacity are considered to be crucial factors. There have been many studies on the employment of compression techniques within the LLC micro-architecture. One facet of the research so far has been the design of compression-aware cache structures, and these structures can be classified into two broad categories: variable-segment caches, and fixed-segment caches. The objective of the variable-segment caches is to maximize the effective capacity of the cache, even at the cost of requiring non-negligible space for remapping and compaction [5, 17]. Different types of data compression into different sizes, so the number of segments used directly impacts how much physical space will be fully utilized for increasing effective capacity. Fixed-segments caches, on the other hand, aim to increase effective capacity without any significant overhead in complexity by fixing the number of segments occupied by the compressed cacheline (usually up to 2 or 4), thereby allowing it to avoid cacheline compaction and manipulation overhead. Since our focus is on reducing the read-hit penalty, the proposed HoPE scheme will work in both variable-segment and fixed-segment cache architectures. However, we will mainly exploit the variable-segment cache architecture for the rest of this work, in order to show results that can be directly compared to the current state-of-the-art in this domain – the Effective Capacity Maximizer (ECM) mechanism [4] – which also employs a variable-segment cache architecture.

3.1.2 Motivation for Early Decompression in Compressed LLCs

Data compression can play an important role, because of its ability to reduce data size. This reduced data size can increase the effective memory capacity, without increasing the physical size of memory, and thus improve memory performance. However, there is a well-known disadvantage that prohibits the use of compression all of the time, as opposed to using compression adaptively and selectively. The disadvantage is the read-hit-decompression overhead [5]. Figure 17 shows a conceptual overview of a compressed memory system using a decoupled variable-segment cache structure. It is mainly composed of compressed memory, a data compressor and a decompressor. For example, if a cacheline is stored with uncompressed data, like U1, it does not need to be decompressed and it can avoid 5 decompression cycles [5]. However, if a cacheline is stored in its compressed form, it must be decompressed before it is read. Figure 18 shows the breakdown of the total execution time of several real application workloads selected from the PARSEC benchmark suite [2]. The benchmarks are executed on a system with a logically 16-way (physically 4-way) compressed LLC using the ECM mechanism [4]. The execution time is normalized to the LRU replacement policy. The details of our evaluation setup are presented in Section 3.4. As shown in Figure 18, a relatively large proportion of execution

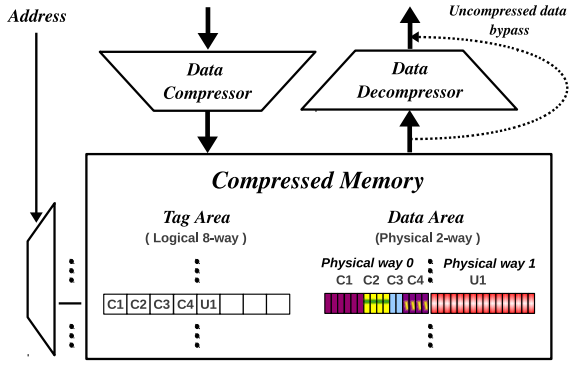


Figure 17: Conceptual overview of a compressed memory system using a decoupled variable-segment cache architecture.

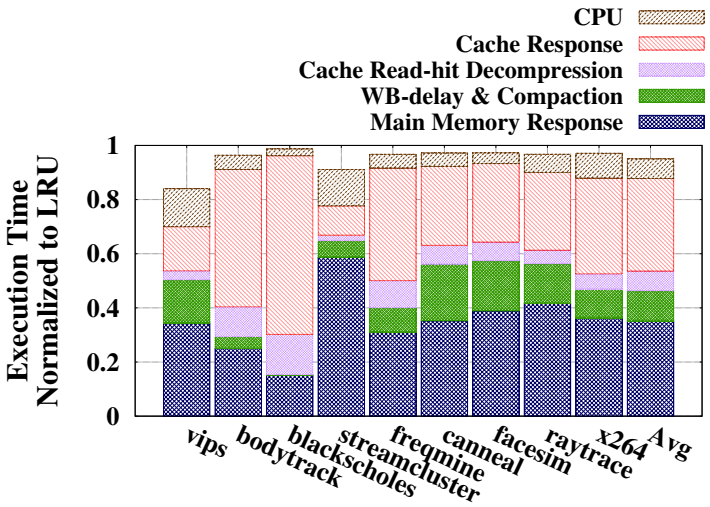


Figure 18: Breakdown of the total execution time of several real application workloads from the PARSEC benchmark suite [2] running on a system using the ECM mechanism [4]. The results are normalized to the LRU replacement policy.

time is spent on read-hit-decompression in several applications, like *bodytrack*, *blackscholes*, and *freqmine*. The read-hit-decompression overhead of *blackscholes* is especially high, coming in at 15.2% of the total execution time. Also, we observed that the percentage read-hit decompression penalty is increased if a policy works particularly well on the compressed LLC. On average, the read-hit-decompression time takes over 7.7% of the entire execution time when employing the ECM policy [4]. Therefore, if one is able to reduce the read-hit-decompression overhead, one can significantly improve the compressed memory performance. The proposed HoPE mechanism adresses precisely this issue; it mitigates the read-hit-decompression overhead with near-negligible hardware overhead.

3.2 Related Work

3.2.1 Adaptive Memory & Cache Compression

3.2.1.1 Software-based Compressed Memory (as used in Apple Inc.'s OS X Mavericks)

Apple Inc. has recently introduced software-managed compressed main memory to its OS X Mavericks operating system, in an attempt to improve responsiveness when the system is heavily loaded [44]. This commercial adoption indicates that the industry is becoming more aware that memory compression is a feasible way to increase performance at low

overhead. Apple Inc. is able to claim performance improvement when a user is running multiple apps, as OS X Mavericks is able to free up space via compression when needed. The company also understands that decompression causes an overhead in reading, so the new operating system adaptively chooses which part of memory to compress via an LRU-like process, resulting in the compression of only idle applications. This principle is similar to our work but it is based on compressing main memory, which is less likely to be near-capacity, and, thus, has less potential for improving system performance. As a side note, compressed main memory has also been previously explored in the industry by IBM Corp.s hardware-oriented Memory Expansion Technology (MXT) [51].

3.2.1.2 Unified Compressed Memory

In the work of [14], the authors propose a unified compressed memory hierarchy that comprises a compressed on-chip LLC and a compressed main memory, modeled after IBMs MXT system [51]. The software-managed compressed cache used, called Indirect Index Cache [15], allocates variable amounts of storage to different cachelines based on their compressibility. As this scheme employs de/compression across two levels of memory hierarchy, it benefits from information transfers in compressed form across the interconnection to realize a bandwidth gain. The idea of decompressing frequently used blocks to reduce decompression overhead is briefly mentioned in [14] and referred to as the "adaptive scheme." However, the approach of this adaptive scheme was to decompress *all* frequently used cache blocks. Since this methodology provided marginal benefits in the evaluated scheme, the concept was not explored any further. This conclusion matches our observations in that we have also found that aggressively decompressing all hot cachelines often degrades performance, as it can significantly reduce the effective capacity. This realization serves as the *primary inspiration of the work proposed in this chapter*: rather than naively decompressing all hot cachelines, we devise a methodology that can **dynamically and intelligently select and eagerly decompress some cachelines, in anticipation of their pending use**. Given that the management of the replacement policies of the cache

architectures in [14] is *software*-managed, it makes a direct and *accurate* comparison with HoPE infeasible. However, we can approximate the adaptive scheme methodology in [14] with a hardware-based design that naively decompresses all hot cachelines (since this is precisely what was performed in [14], as previously mentioned). We approximate this using a hardware-based DRRIP scheme, augmented with the capability to decompress all hot cachelines. In this way, we will provide a comparison of this approach with all the other designs in Section 3.4.3.

3.2.1.3 Adaptive Cache Compression

There has been previous research on using adaptive cache compression in compressed LLC systems [5]. The work in [5] observed the read-hit overhead in using compressed LLC systems and, thus, proposed to use a 19-bit global counter of penalized hits vs. avoidable misses (based on the LRU stack depth) to decide whether or not to decompress at that time period. The scheme works by essentially compressing all subsequent cachelines when memory is under load, and leaving all subsequent cachelines uncompressed when memory is not under significant load. The scheme is able to perform as well as full compression when the workload is memory-intensive, and as well as no compression when the workload is not memory-intensive. However, this technique is somewhat orthogonal to our idea, because the mechanism in [5] chooses when to leave a cache fully compressed, or fully uncompressed, whereas the proposed HoPE framework assumes a fully compressed cache, and then picks certain cachelines to leave uncompressed.

3.2.2 Compression-aware Replacement Policies

Another area of active research is is in the cacheline replacement policies, as they have a significant impact on system performance. Extensive research has been conducted in developing efficient cache management policies to take advantage of locality and cache access patterns [1, 28, 29, 30, 39]. However, in a compressed cache, the compressed *size* information can also provide a good hint in finding a victim cacheline (in addition to the

locality information). One recent study proposed a compressed cache replacement policy, called the Effective Capacity Maximizer (ECM) [4], which aims to balance the locality and size information in the cache management policies. Since this technique is currently the best-performing replacement policy in compressed cache systems, we briefly explain the overall mechanism in this sub-section.

As shown in Figure 19, ECM extends the Re-Reference Interval Prediction (RRIP) framework [1] – which only considers locality information – with the ability to also consider the cacheline size information. The operation of RRIP revolves around the use of 2^M possible states (Re-Reference Prediction Values, RRPV), which correspond to a prediction of when a cacheline is likely to be re-referenced. For example, if a cacheline is in the state of RRPV= $2^M - 1$, it is predicted that the cacheline will be re-referenced in the *near-immediate* future. If a cacheline is in the state of RRPV=0, it is predicted that the cacheline will be re-referenced in the *distant* future. Therefore, RRIP considers cachelines with state RRPV= 0 as least likely to be re-referenced, and so it classifies cachelines in the RRPV= 0 state as the eviction pool. If there are no cachelines to evict in the RRPV= 0 pool, RRIP demotes all of the cachelines' RRPV states. Conversely, on a hit, cachelines are promoted to higher RRPV values, as they are predicted to be more likely to be referenced. Cachelines are normally inserted at RRPV=1, but such a static insertion scheme can cause memory thrashing when the working set is just greater than the available memory. To address this problem, the DRRIP extension was also developed, which dynamically decides between using a static insertion scheme, or a bimodal insertion scheme that is better suited to handling cache thrashing [1].

ECM [4] demonstrated that the size information is also useful in increasing the effective capacity of compressed caches, and, thus, it uses this information to determine insertion and eviction policies. The ideas proposed in ECM are fundamentally different from those in the newly proposed HoPE framework; the latter prioritizes *locality* information to reduce compression overhead. However, the ECM architecture still provides key insight on how to

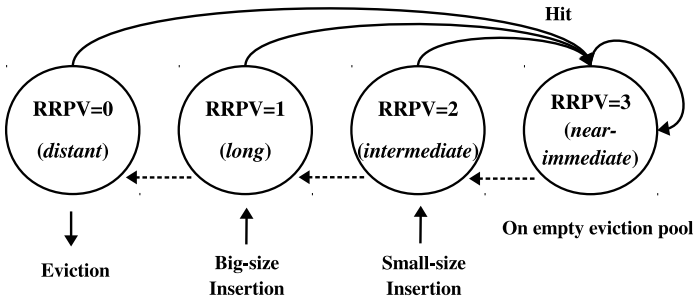


Figure 19: The ECM cache management framework [4] with a 2-bit (M=2) Re-Reference Prediction Value (RRPV).

also take into account *size* information. In ECM, after classifying a cacheline as either big-size or small-size, ECM's Size-Aware Insertion (SAI) policy sets the small-size cachelines' initial RRPV as 2, while it sets the big-size cachelines' initial RRPV as 1, in order to give the big-size cachelines a higher chance of eviction. Furthermore, ECM's Size-Aware Replacement (SAR) policy selects the biggest-size cachelines among the cachelines whose RRPV is 0 as victims, so that the cache can accommodate more small-size cachelines (thereby increasing the effective capacity of the cache). The performance improvements shown later on are often normalized to the ECM scheme, as it is currently the state-of-the-art in compression-aware replacement policies.

3.3 The Hot-cacheline Prediction for Early Decompression (HoPE) Framework

The proposed framework mainly aims to reduce the read-hit penalty. This goal is achieved by speculatively decompressing frequently used cachelines. However, in decompressing cachelines, we must evict one, or more, other cachelines, in order to accommodate the new uncompressed cacheline. This results in a reduction in the effective capacity, despite the increase in read-hit performance. In order to balance the effective cache capacity and the read-hit performance, we take into consideration not only the locality, but also the size information of the compressed cacheline. The reconciliation of the two conflicting characteristics of decreasing effective cache capacity when improving read-hit performance is

achieved using the proposed HoPE framework. The new mechanism relies on three constituent fundamental policies, which work synergistically and in unison: (1) *Hot-cacheline Prediction (HP)*, (2) *Early Decompression (ED)*, and (3) *Hit-history-Based Insertion (HBI)*. Specifically, the HP and ED policies determine when a compressed cacheline should be decompressed to minimize the total execution time with minimal hardware cost. Additionally, we observe that cachelines of similar compressibility often have a high correlation in their hit rates, as data with similar structures are often used in similar ways. Therefore, the third policy – HBI – takes advantage of this correlation by predicting how often a cacheline will be hit, based on its compressibility attributes.

3.3.1 The Hot-cacheline Prediction (HP) Technique

To reduce the read-hit-decompression overhead, we must first find frequently used cachelines to decompress. The simplest way to find a frequently used cacheline is to use a counter for each cacheline, in order to determine frequency of use. However, if we use a separate counter for each cacheline, there will be significant storage overhead and additional complexity in comparing these counters. To reduce this storage overhead and comparison operations, we propose the Hot-cacheline Prediction (HP) framework to find hot cachelines more efficiently.

The main concept of the proposed HP technique is the hybridization of the locality-aware replacement policy framework with the hot-cacheline prediction framework, so that we can use the framework for both replacement and hot cacheline decisions. To accomplish this task, we extend the RRIP framework, since, interestingly enough, the method for finding hot cachelines can be viewed as an extension of the method used in finding cachelines with high locality. Figure 20 shows the proposed multi-purpose HP re-reference chain, with both locality and hot-cacheline counting states.

For the locality context, we use half of the possible states (from *distant* to *near-immediate*), plus one additional state (*immediate*), to represent locality information in a manner similar to RRIP and ECM. However, in our proposed HP chain, we then use the rest of the states to

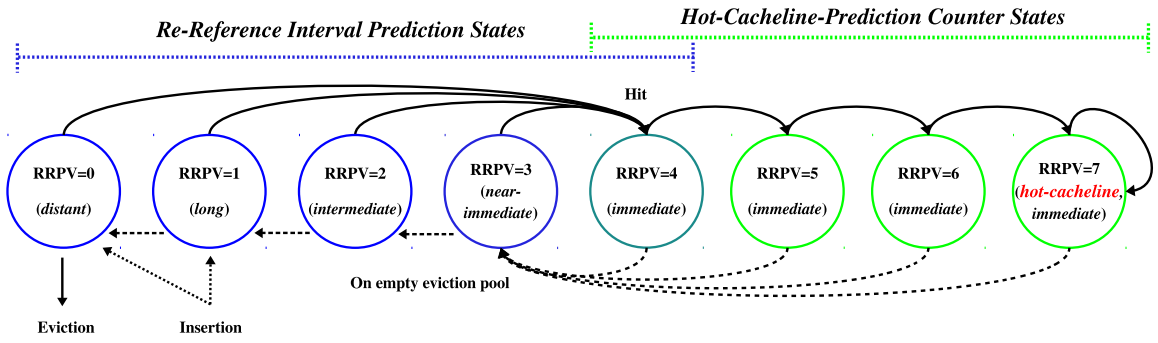


Figure 20: The proposed multi-purpose Hot-cacheline Prediction (HP) chain, with its locality and hot-cacheline-counting states, assuming a 3-bit RRPV.

represent counting states for cachelines in the *immediate* state. This works by incrementing the state when a cacheline in the *immediate* state is hit again. If a cacheline gets hit after arriving at the RRPV= 7 state, the cacheline is classified as a hot cacheline and then decompressed. In other words, if a compressed cacheline gets five consecutive hits, it will be decompressed. If a miss occurs on the cacheline and there are no more cachelines in the eviction pool, however, the cacheline will be demoted to the original *immediate* state.

Clearly, there is very little storage overhead in using this hybrid technique. Moreover, each additional bit added to the M-bit RRPV HP chain will double the number of locality and counter states at negligible additional hardware cost.

3.3.2 Enabling Early Decompression (ED)

In this sub-section, we propose two Early Decompression (ED) schemes that employ the previously introduced HP mechanism. After we classify cachelines as hot cachelines, a simplistic ED policy could speculatively decompress all of the hot cachelines to reduce the read-hit-penalty. However, decompressing a cacheline would imply the eviction of one or more (up to eight) cachelines to fit the newly decompressed cacheline. Decompressing without considering the hot cacheline's size will reduce the effective cache size and may, at times, degrade performance. Thus, we propose static and dynamic threshold schemes to determine which cachelines we can decompress without significantly affecting the effective

cache size.

3.3.2.1 Static Early Decompression (SED)

The SED scheme selectively chooses to decompress hot cachelines whose compressed sizes are larger than a statically-set threshold. This method works on the principle that decompressing a large compressed cacheline requires fewer number of additional segments to store the uncompressed cacheline than decompressing a smaller compressed cacheline. The threshold for SED is set based on the number of segments a compressed cacheline can occupy. Because cachelines can be compressed to anywhere from 2 segments (the most compressible) to 16 segments (uncompressed), we can statically set the threshold for SED between these values, $2 \leq T_h \leq 15$. Note that a threshold value of 16 would amount to the degenerate case of not decompressing.

3.3.2.2 Dynamic Early Decompression (DED)

Even though we may be able to tune SED to find a well-performing threshold per application, the static scheme can never be really robust, since the best-performing threshold varies not only with the application, but also with the execution phase of the application [4]. With that in mind, we introduce the DED scheme, where we adjust the threshold size dynamically, based on the effective cache capacity and the physical usage of the memory. We develop the DED scheme by adapting the Dynamically Adjustable Threshold Scheme (DATS), in Section 2.3.2.2, to decide suitable threshold sizes that would trigger early decompression. The DED aims to determine an appropriate threshold that prevents the situation where too many small-size cachelines are evicted by a decompressed cacheline. When there are more small-size cachelines, the threshold becomes larger, so that the probability of small-size cachelines being evicted is smaller. Equation 1 below shows how the dynamically changing threshold is determined by the DED mechanism.

$$T_h = \left\lceil \left(\left(\frac{NS_{uc} \times W_P}{W_L} \right) - \left[NS_{uc} \times \left(1 - \frac{NT_v}{W_L} \right) \right] \right) \times \left(\frac{Size_{total}}{NS_{uc} \times W_P} \right) \right\rceil \quad (1)$$

The term NS_{uc} is the number of segments used for an uncompressed cacheline. The terms W_P and W_L indicate the physical and logical number of ways, respectively. In this study, we set $NS_{uc} = 16$, $W_P = 4$, and $W_L = 16$. When a new cache line arrives, two variables (NT_v and $Size_{total}$) change. The term NT_v is the number of valid tags within a set, which indicates the *effective capacity*, while $Size_{total}$ is the total number of valid segments in the data area of a set, which indicates the *physical memory usage*. The left-most term (($NS_{uc} \times W_P$) / W_L) represents the average number of segments in a logical way. The right-most term in the equation above ($Size_{total}$ / ($NS_{uc} \times W_P$)) represents the physical memory usage ratio. Thus, the threshold (T_h) increases dynamically with the effective capacity (NT_v), while other terms adjust the threshold within a valid range.

3.3.3 The Hit-history-Based Insertion (HBI) Policy

The ECM mechanism prioritizes size information over locality information, because it aims to increase the effective cache capacity by evicting larger-sized cachelines [4]. However, the proposed HP policy considers locality information as more important than size information, because identifying hot cachelines with high hit rates improves performance. To that end, we observe that cachelines of similar compressibility often have a high correlation in their hit rates, since data with similar structures are often used in similar ways. To take advantage of this correlation, we propose the HBI scheme, as a means to predict how often a cacheline will be used, based on its compressibility.

We analyze the correlation between locality and size information by keeping track of the hit rate of each cacheline size. Because hit rates fluctuate over an application's execution time, it is important to observe this time dependence. Figure 21 illustrates an example hit-history trace for each compressible cacheline size over a period of 400 million cycles in the execution of the *facesim* application [2]. We assume the use of a 10-bit Hit-history Global Counter Table (HGCT) to keep track of the hit rates of each possible cacheline size. Figure 22 shows a high-level overview of the HGCT structure employed by the HBI scheme. The HGCT keeps track of the hit rates of cachelines of similar compressibility with a global

counter for each compressibility bin. Using the HGCT, we are able to predict the likelihood that cachelines of a certain size will be hit. The HGCT is updated on each hit, or miss, of a particular cacheline size; a hit increments the counter, while a miss decrements the counter. Figure 21 shows the global hit counter value for each cacheline size. In this graph, the consistently high/low values indicate that the particular cacheline size is consistently hit, or missed, in that time interval. Similarly, quick-changing lines from 0–1024 or 1024–0 also indicate periods of consistent hits, or misses. From this graph, it is possible to see that similarly-sized cachelines can often exhibit similar hit/miss rates. Additionally, for most cases, the changes from consistent hits to consistent misses are abrupt. Using this scheme, we are, therefore, able to predict the hit-rate performance of each cacheline size and use it to guide our insertion policy.

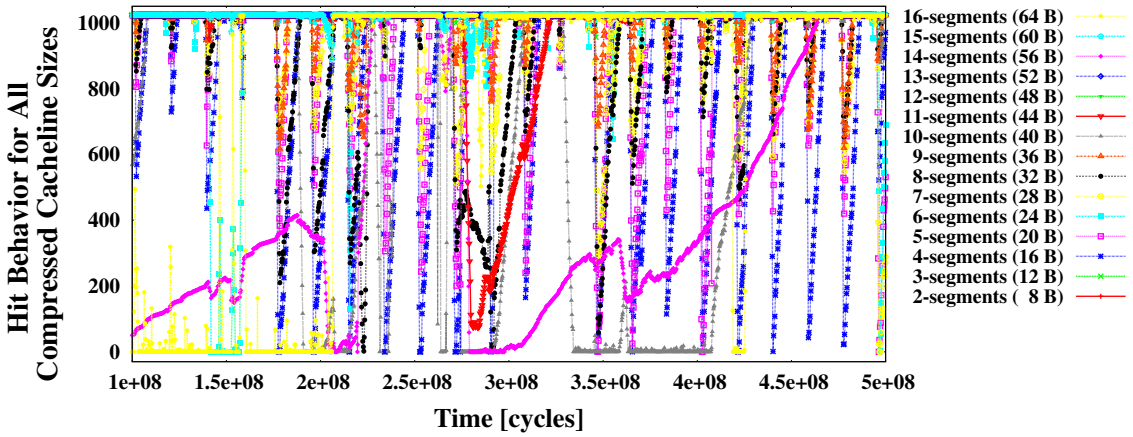


Figure 21: The hit-history traces for each compressible cacheline size over a period of 400 million cycles in the execution of the *facesim* application [2]. We assume the use of a 10-bit Hit-history Global Counter Table (HGCT) to keep track of the hit rates of each possible cacheline size. The structure of the HGTC is shown in Figure 22.

We therefore introduce the notion of hit-history-based insertion (i.e., the HBI policy) using the HGCT structure of Figure 22. When inserting a new cacheline of a particular size, we use the most significant bit (MSB) of the corresponding HGCT counter to determine whether or not the cacheline is likely to be hit in the near future. If the MSB is 1, it denotes that – based on the hit history – the new cacheline will more likely be referenced frequently,

so it will be placed in a higher RRPV= 1 state. Conversely, if the MSB is 0, it denotes that the new cacheline is more likely to have low locality and will, thus, be placed in a lower RRPV= 0 state.

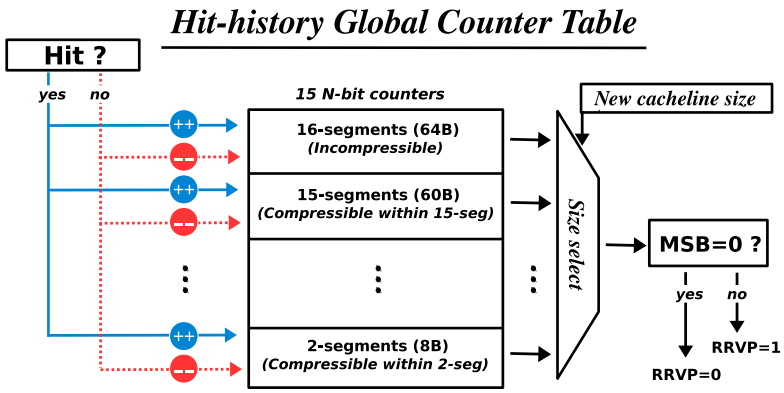


Figure 22: High-level overview of the N-bit Hit-history Global Counter Table (HGCT) structure employed by the HBI scheme. The HGCT keeps track of the hit rates of cachelines of similar compressibility with an N-bit global counter for each compressibility bin.

3.4 Experimental Evaluation

3.4.1 Methodology

3.4.1.1 Simulation Framework

We built a trace-driven simulator to evaluate the proposed HoPE mechanism. Memory access sequences and compression ratios are the most important factors for our evaluation, so we have obtained our traces using the Simics [40] full-system simulator, augmented with GEMS [41]. We obtain traces by running nine multi-threaded benchmarks from the PARSEC benchmark suite [2] for 300 million instructions. We simulate a quad-core processor with a two-level cache hierarchy, with the L1 caches using an LRU replacement policy. Three different LLC management schemes were evaluated and compared: (1) the **DRRIP scheme [1]**, (2) the **ECM mechanism [4]**, and (3) the proposed **HoPE framework**. We also present a performance comparison against the **approximated adaptive scheme of [14]** (see Section 3.2.1.2 for details), which we call **AAS** (for Approximated Adaptive Scheme). In most cases, the experiments assume the conventional **LRU** policy as a reference point.

The details of our simulation parameters are described in Table 3.

We also evaluated the projected energy consumption of the proposed HoPE framework. The energy parameters of the SRAM-based LLC and the DRAM-based main memory system are obtained from CACTI [42] and a commercial DRAM data sheet [43], respectively.

3.4.1.2 The Compression Technique Employed in the LLC

We target a compressed LLC architecture in this work. No compression is assumed in the L1 caches and main memory. The compression overhead far outweighs any benefits from compression in L1, while most applications can fit in main memory without any need for compression. All schemes assume a compressed LLC. We use Frequent Pattern Compression (FPC) – a bit-level compression algorithm – in this work, as it provides high compression performance at a low overhead, in terms of both delay and implementation cost. The energy parameters of the FPC compression algorithm were obtained from [16]. We apply this FPC per word, which is able to regularly achieve maximal compression of 64 B cachelines (comprising 16 decoupled segments) down to 8 B sizes (comprising only 2 decoupled segments), as will be shown in the next section. A 2 MB physically 4-way LLC is the baseline LLC architecture. Using a compressed LLC architecture, we are able to increase the logical ways in a set up to 16, which can increase the effective LLC capacity up to 8 MB, i.e., a significant improvement with low area overhead.

Table 3: Simulated system configuration details for evaluating HoPE.

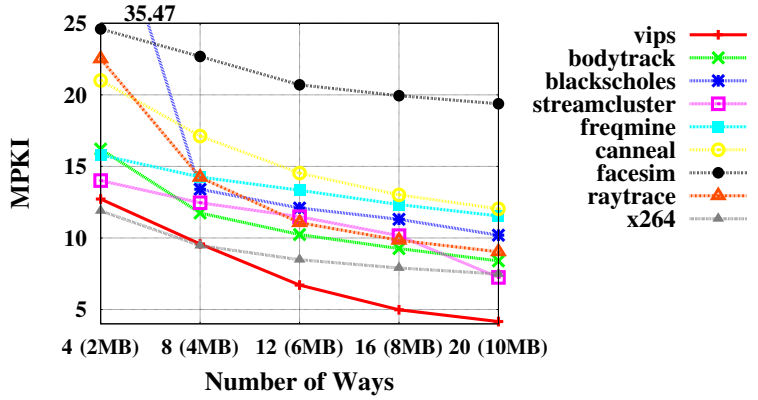
Number of CMP Cores	4
Processor Core Type	UltraSPARC-III+, 2 GHz
L1 caches (private)	I- and D-caches 32 KB, 4-way, 64 B
L1 response latency	3 cycles
L2 caches (shared)	2 MB, 4-way, 64 B, NUCA, MESI
L2 response latency	20 cycles
L2 read-hit overhead	5 cycles for decompression
L2 writeback buffer	8 entries
Compaction overhead	16 cycles
DRAM memory	DDR2 4 GB
Memory response latency	450 cycles

In a variable-segment compressed LLC, incoming requests (read misses, write hits, and write misses) require contiguous segments to store the new data. Thus, data compaction must be used to free contiguous cache segments. To account for such overhead, we analyze when compaction overhead can be hidden and when it must be factored into the delay. In the case of read misses, the compaction delay can be hidden, since compaction can be performed while the cache is waiting on the data from main memory. However, in the case of write hits and misses, the updated cacheline size is likely to be different from the entry, or entries, it replaces, so – in order to free up contiguous space for writes – compaction must be performed immediately. This causes overhead. Hence, we include write miss/hit compaction delays in our evaluations to account for compaction overhead, as shown in Table 3.

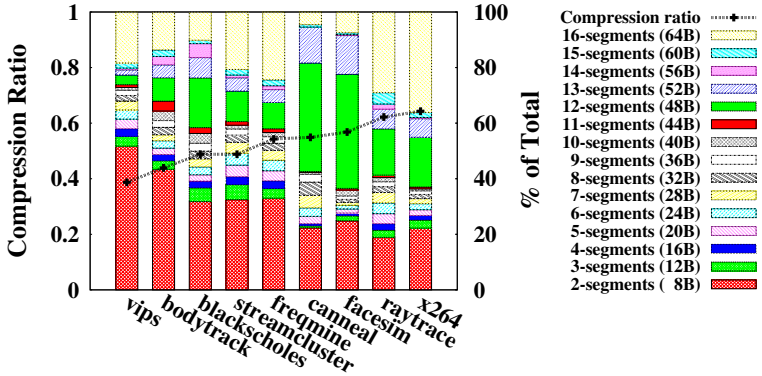
3.4.2 Workload Characteristics

In order to better understand the results, we first analyze several workload characteristics relevant to the proposed HoPE scheme. The pertinent characteristics to note are the set-associativity of the cache, the compressibility of workloads, and the locality performance of each cacheline size. In terms of set-associativity, we assume a logically 16-way cache overlaid on a physically 4-way 2 MB LLC using compression, which increases the maximum effective capacity to 8 MB. This is, in effect, similar to increasing the number of ways of the cache, so we find it appropriate to show how sensitive our workloads are to the way-associativity, as shown in Figure 23(a). In particular, the sensitivity of applications to the effective capacity guides our decompression schemes, since we trade effective capacity for better read-hit performance. The x-axis of the figure shows way-associativity, while the y-axis shows misses per thousand instructions (MPKI). The MPKI of *blackscholes*, in particular, shows the highest sensitivity to the way-associativity. In terms of compressibility, we show in Figure 23(b) the average compression ratio and proportion of each compressed cacheline size. The figure shows that workload data can be compressed to 38.7%–64.3% of its original size, on average, with *vips* in particular enjoying compressibility to 38.7%

of its original size, and having 51.9% of its total cachelines compressible to 2 segments (8 B). Additionally, we find that cachelines of similar compressibility often exhibit similar hit performance, as shown in the example hit-history trace in Figure 21. Thus, in out design, we take into consideration both the locality and the compressibility, and their correlation with respect to time.



(a) Miss behavior sensitivity to physical cache set-associativity

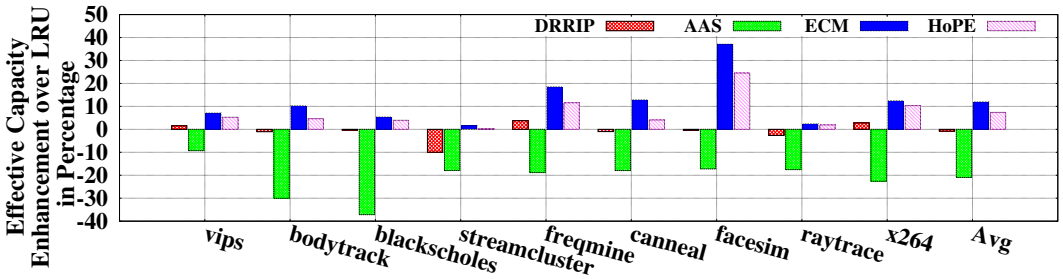


(b) Achievable compression ratios and distributions of line sizes

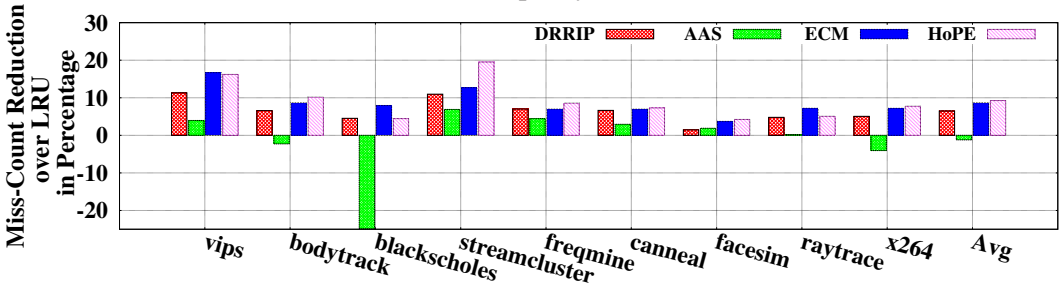
Figure 23: The salient cache-related characteristics and behavior of the application workloads used in this study.

3.4.3 Performance Evaluation of the HoPE Framework

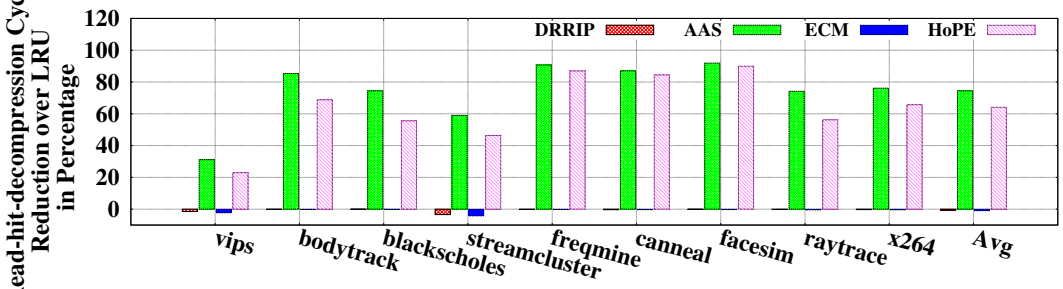
Detailed evaluation of the constituent policies of our scheme will be explained in subsequent sections. Here, we show the overall performance benefits obtainable by using the proposed HoPE scheme. Figure 24 shows overall performance results of compressed LLCs using the DRRIP [1], AAS [14] (see Section 3.2.1.2 and Section 3.4.1.1), ECM [4], and



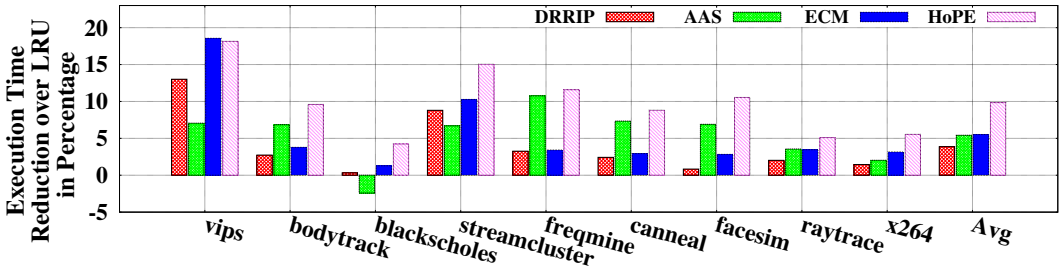
(a) Effective Capacity Enhancement



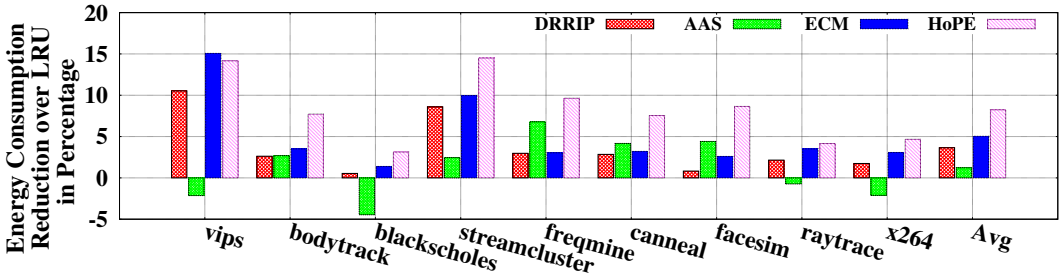
(b) Miss-Count Reduction



(c) Read-hit Penalty Reduction



(d) System Performance Improvement



(e) Energy Consumption Reduction

Figure 24: Overall performance evaluation of DRRIP [1], ECM [4], and the proposed HoPE framework, across a number of key metrics. All results are normalized to the performance of LRU.

HoPE designs – normalized to the performance of LRU – for all evaluated applications. The DRRIP, AAS, ECM, and HoPE designs employ 4-bit RRPVs for fair comparison, and a 2-bit HGCT is used in the HBI policy of HoPE. As mentioned in Section 3.2, the adaptive scheme of [14] is approximated by augmenting DRRIP with the ability to decompress all hot cachelines. Note that the sensitivity to the size of the RRPV register and the size of the HGCT counters will be explored in subsequent sections. Figures 24(a) and (b) show the effective capacity enhancement and the miss-count reduction, respectively. The applications *streamcluster*, *freqmine*, and *facesim* exhibit reductions in the miss count by utilizing the HBI policy of HoPE, but, on average, the proposed HoPE mechanism shows lower reduction in the miss count than ECM, due to the decrease in the effective capacity by pre-decompressing hot-cachelines. The application *blackscholes*, in particular, experiences an increase in the number of misses, because it is very sensitive to the effective capacity, as explained in Section 3.4.2. However, HoPE achieves better cache performance improvement than DRRIP, on average.

Although the proposed HoPE framework shows an increase in the miss count by decompressing hot cachelines, it also shows a significant reduction in the read-hit penalty, as shown in Figure 24(c). All applications benefit from HoPE in terms of read-hit decompression penalty reduction – the *freqmine*, *canneal*, and *facesim* applications, in particular, show over 80% reduction. Even though AAC shows greater reduction in read-hit penalty over HoPE, it also shows a significant increase in the miss count, as shown in Figure 24(b), because of the decrease in the effective capacity resulting from the pre-decompression of hot cachelines without considering the size information. By reducing the read-hit decompression penalty while considering the size information with DED, the proposed HoPE scheme reduces the execution time by 9.8% over LRU, 6.2% over DRRIP, 4.7% over AAS, and 4.6% over ECM, on average, as shown in Figure 24(d). Note that only the *vips* application shows some performance degradation when compared to ECM, but if we choose a *vips*-specific optimal static threshold, we can still improve the performance of *vips* relative

to ECM, as will be shown later on (in Figure 26 (b)). Finally, the reduced execution time and reduced number of decompressions when reading from the LLC enables us to reduce the energy consumption by 8.3% over LRU, 4.8% over DRRIP, 8.7% over AAS, and 3.4% over ECM, on average, as shown in Figure 24(e).

3.4.4 Dissecting the Key Attributes of HoPE

3.4.4.1 Dynamic vs. Static Early Decompression

The benefits of the HoPE framework emanate from the use of eager decompression. Here, we evaluate the performance of the *dynamic* (DED) and *static* (SED) early decompression schemes, and we compare them against the ECM mechanism [4], which does not employ any form of early decompression.

Starting with the static decompression scheme, we find that static threshold schemes work well, provided optimum threshold values can be found a priori. Figure 25 shows the percentage difference in execution time when using the HP+SED setup (i.e., the Hot-cacheline Prediction and Static Early Decompression policies combined) with various SED threshold values, as compared to the baseline ECM. Note that the threshold value Th=2 provides some interesting results, since this scheme even decompresses 2-segment-sized cachelines, which comprise over 20% of the cachelines seen in each benchmark and require significant (64-8=56 B) additional space to be stored uncompressed. Due to the high proportion of 2-segment-sized cachelines, performance often varies significantly between the SED Th=3 and Th=2 cases. This is exemplified in the case of *vips* and *blackscholes*, whose performance degrades significantly in the Th=2 case. The application *vips* is sensitive to the Th=2 threshold, since its 2-segment cachelines comprise over 53% of all cachelines; *blackscholes* is also sensitive to this threshold, because its performance is very sensitive to the effective cache capacity, as shown in Figure 23(a). However, the static threshold Th=2 yields the best performance for *bodytrack*, *freqmine*, *canneal*, and *facesim*. For these applications, decompressing the most compressible cachelines is still worthwhile as those small cachelines often show relatively high hit-rates. As shown in Figure 25, the best performing

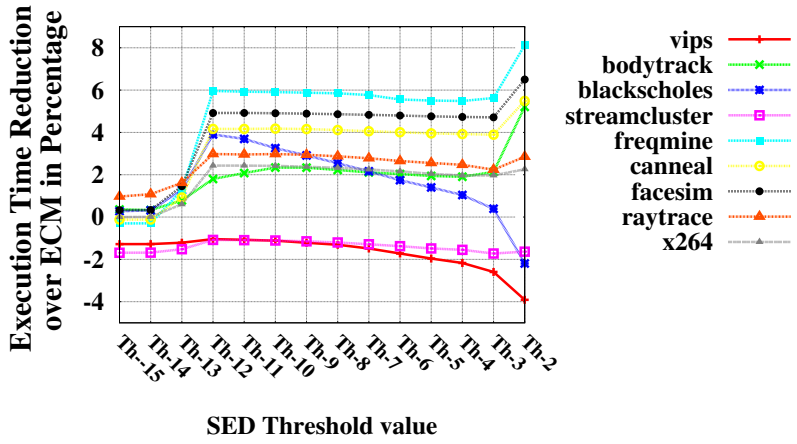
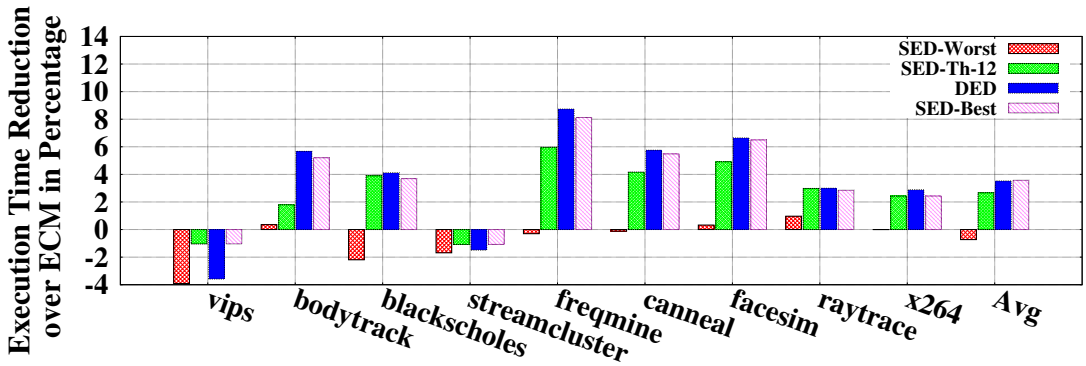


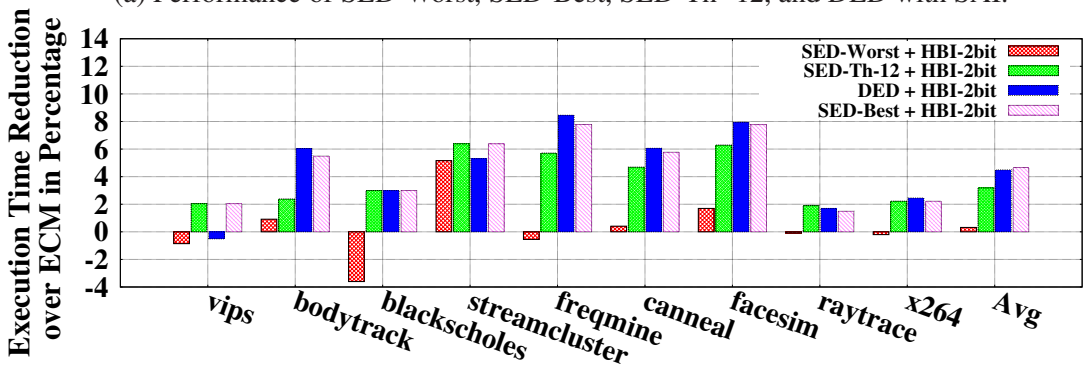
Figure 25: Percent difference in execution time when using the HP+SED setup (i.e., the Hot-cacheline Prediction and Static Early Decompression policies combined) with various SED threshold values, as compared to the baseline ECM [4].

threshold value is very application-dependent, though the threshold Th=12 shows the most reliable performance, on average, by conservatively pre-decompressing only big-sized hot cachelines.

Continuing with the dynamic decompression scheme, we find that the dynamic threshold scheme is able to compensate for application-specific behavior, by adjusting the threshold based on physical memory usage and effective capacity. We find that the dynamic threshold scheme not only gives more reliable performance, on average, but it often beats the best static threshold, since application behavior changes over the execution time. Figure 26 shows the percentage difference in execution time over ECM, when using the HP policy with SED-Worst, SED-Best, SED-Th=12, and the DED setups with either SAI (as used in ECM), or a 2-bit HBI. The SED-Worst and SED-Best values are selected among the SED threshold values T_h, where $2 \leq T_h \leq 15$, for each application. Note that the threshold value – for the SED-best and SED-worst results – varies with each application. As shown in Figures 26 (a) and (b), the DED policy, which dynamically changes the threshold for early decompression, shows the best performance, except for the *vips* and *streamcluster* applications. On average (over all benchmarks), DED shows a 4.1% reduction in execution time over SED-Worst, while SED-Best shows only a 0.2% reduction in execution time over



(a) Performance of SED-Worst, SED-Best, SED-Th=12, and DED with SAI.



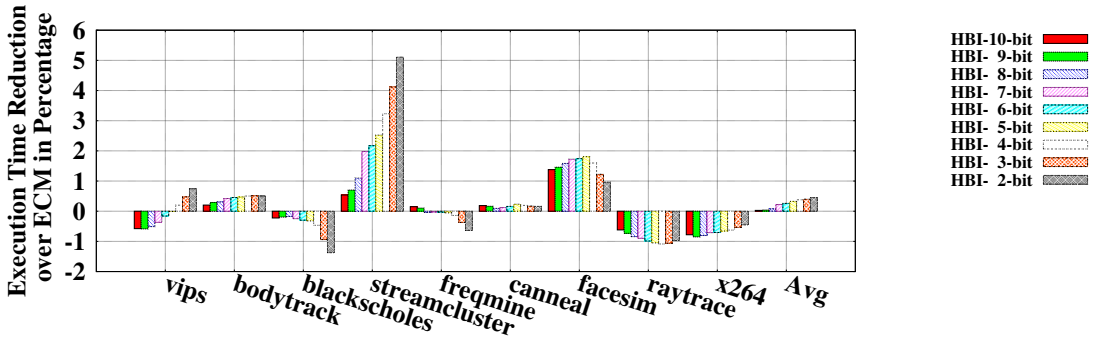
(b) Performance of SED-Worst, SED-Best, SED-Th=12, and DED with a 2-bit HBI.

Figure 26: Percent difference in execution time over ECM [4], when using the HP policy with SED-Worst, SED-Best, SED-Th=12, and the DED (i.e., Dynamic Early Decompression) setups with either SAI (as used in ECM), or a 2-bit HBI.

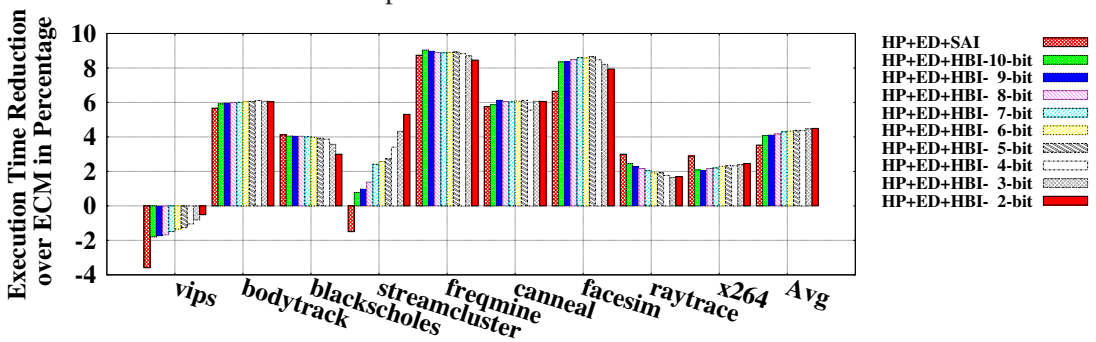
DED (when a 2-bit HBI is used), as shown in Figure 26(b). Though DED does not perform better than SED-Best, it performs exceptionally well in that it is able to improve system performance significantly, without the need to determine optimal thresholds a priori.

3.4.4.2 Comparing HoPE's Hit-history-Based Insertion (HBI) to ECM's Size-Aware Insertion (SAI) [4]

The primary objective of this work is to reduce the read-hit penalty by finding hot cache-lines to preemptively decompress. This differs fundamentally from the ideas proposed in ECM [4], as the proposed decompression scheme optimizes performance by focusing on locality. The Size-Aware-Insertion (SAI) policy proposed in ECM prioritizes size information when performing insertions into the re-reference chain. Even though the policy has been demonstrated to improve the effective capacity, we provide a more suitable alternative that prioritizes using the locality information (namely, the HBI policy), which can yield



(a) Performance of HBI with different number of HGCT counter bits (ranging from 2 to 10), as compared to the baseline ECM's SAI.



(b) Performance of the combined HoPE framework (i.e., HP+ED+HBI) against a setup combining SAI+HP+ED, normalized to ECM.

Figure 27: Percent difference in execution time when using the proposed HBI policy, as compared to the SAI policy in ECM [4]. The results are normalized to the baseline ECM.

better results.

Figure 27(a) shows the percentage difference in execution time when using the proposed HBI policy with different number of HGCT counter bits (ranging from 2 to 10), as compared to the SAI policy in ECM. As shown in the figure, the HBI scheme shows performance improvement in the *bodytrack*, *streamcluster*, and *freqmine* applications. In particular, *streamcluster* shows good compatibility with the 2-bit HBI by quickly adapting to the hit-history. Depending on the application, different number of bits for the hit-history are preferred, since different applications show different performance improvement, based on the counter bits. On average, the 2-bit HBI scheme shows the best performance improvement. Even though the HBI scheme by itself shows little performance improvement over SAI, it shows more appreciable improvement when we combine it with the HP+ED policies

(i.e., forming the complete HoPE framework), as shown in Figure 27(b). For fair comparison, the HP+ED policies have also been added to ECM's SAI policy. The locality-aware HBI policy helps the HP and ED techniques find the hot cachelines faster. In other words, HBI helps filter cachelines for locality instead of size, causing the cache to be filled with larger, more "local" cachelines, which are suitable for HP and ED. As a result, the combination of the proposed HBI policy with the HP and ED policies (HBI+HP+ED) shows 4.5% system performance improvement over the baseline ECM, while the SAI+HP+ED setup shows 3.5% improvement. These results highlight the synergistic nature of the three constituent policies comprising the HoPE framework.

3.4.5 Hardware Overhead Analysis

Similar to DRRIP [1] and ECM [4], the HoPE framework requires an M-bit RRPV register per cache block, as shown in Table 4. Additionally, it needs extra counters for the Hit-history Global Counter Table (HGCT) shown in Figure 22. This overhead scales linearly with the number of possible compression sizes used in the compressed LLC architecture. For example, in this work, we assume one 2-bit counter (i.e., N=2 in Figure 22) for each of 15 possible compression sizes (from the most compressible 2-segment cachelines to the uncompressed 16-segment cachelines). This would require 2×15=30 extra bits and a 4-bit multiplexer to make insertion decisions. Obviously, such overhead is near-negligible when compared to the overall size of the cache.

Table 4: Overview of hardware overhead for HoPE.

Replacement Policy	Hardware Overhead[1]
LRU	$n \log_2 n$
DRRIP, ECM, and HoPE	$M \times n$

[1]The number of **extra bits** required **per cache set** in a **logically n-way set associative cache**.

3.4.6 Sensitivity Analysis of HoPE to Various Parameters

Certain design parameters can affect the system performance achieved when using the proposed HoPE framework. In this section, we show how performance is impacted when varying several key design parameters.

3.4.6.1 Sensitivity to the Size of the (M-bit) RRPV Register

Since the size (in bits) of the RRPV register – as used in the HP policy – affects both the time to learn the locality information and the efficacy of the counter states (for hot-cacheline prediction), it is one of the key design parameters in the proposed HoPE mechanism. Figure 28 shows the sensitivity of HoPE to the size of the RRPV register. The results are averaged across all of the tested applications, and values are normalized to LRU. As shown in this figure, the performance improvement of HoPE starts to saturate at 4 bits, indicating that more counter states are not helpful in further reducing the execution time. As expected, a 2-bit register is shown to be insufficient in learning the locality information.

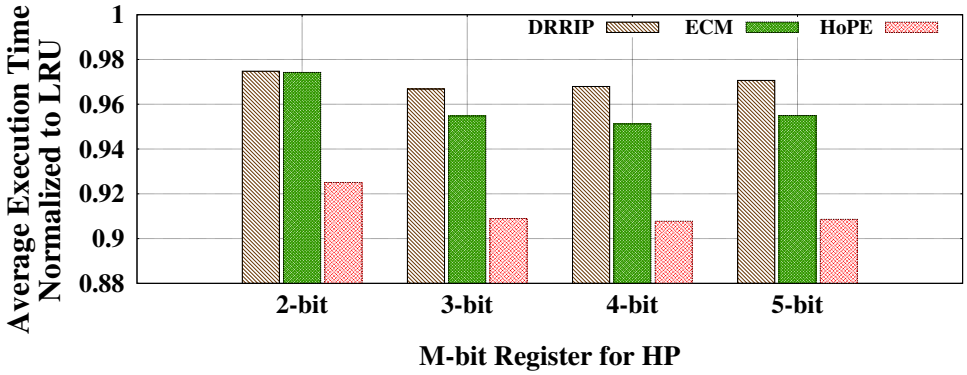


Figure 28: Performance sensitivity of HoPE to the size of the M-bit RRPV register (used in the HP policy).

3.4.6.2 Sensitivity to the Cache Size

Figure 29 compares the average execution time when varying the physical LLC size, from 1 MB to 16 MB. All the results are normalized to a 1 MB LLC with LRU. As shown in the figure, the proposed HoPE scheme shows performance improvement under every physical LLC size, indicating that HoPE is scalable with the physical LLC size. Specifically, HoPE

outperforms ECM by 1.9%–5.3% for various cache sizes.

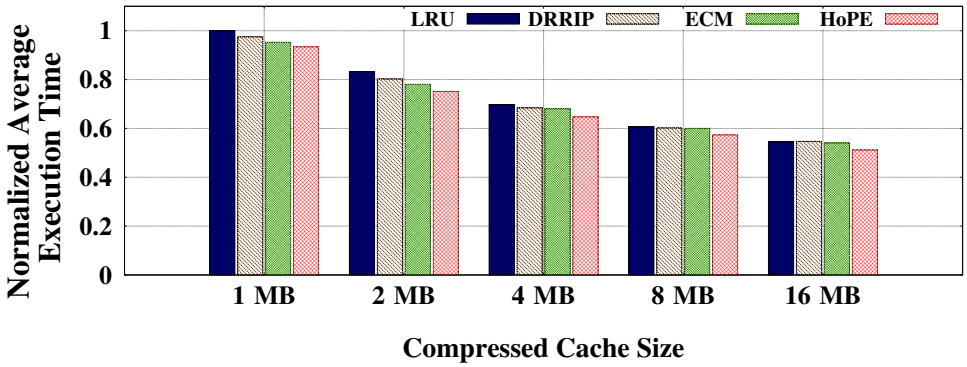


Figure 29: Performance sensitivity of HoPE to the physical LLC size.

3.4.6.3 Sensitivity to the Set Associativity

The sensitivity of HoPE to the number of logical ways was also evaluated. Figure 30 shows results with 8-way, 12-way, 16-way, and 20-way logical set associativities. All the results are normalized to an 8-way LRU. The physical size of the LLC is 2 MB in all designs under evaluation. Clearly, the HoPE framework is scalable with the logical set associativity. Specifically, HoPE outperforms ECM by 3.9%–4.8% for various set associativities.

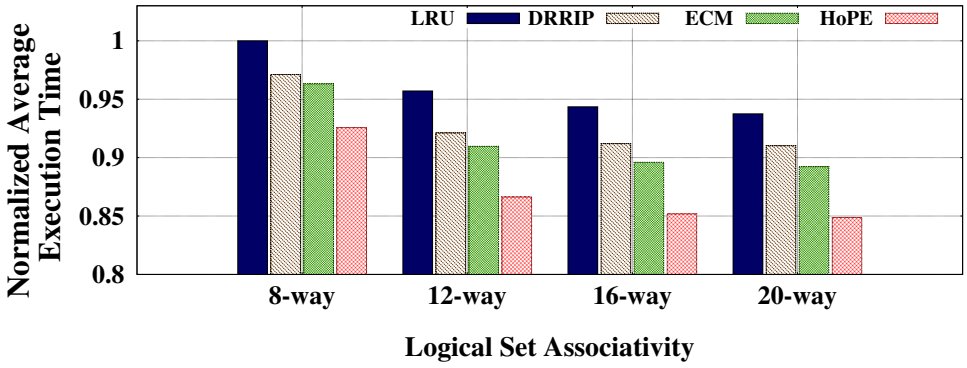


Figure 30: Performance sensitivity of HoPE to the set associativity. A physical LLC size of 2 MB is assumed.

3.4.6.4 Sensitivity to the Decompression Latency

Different compression algorithms have different decompression latencies. Here, we evaluate the performance sensitivity of the proposed HoPE design to the assumed decompression

latency, as illustrated in Figure 31. All the results are normalized to LRU for each respective decompression latency. As shown in the figure, HoPE is able to yield performance improvements across all evaluated decompression latencies.

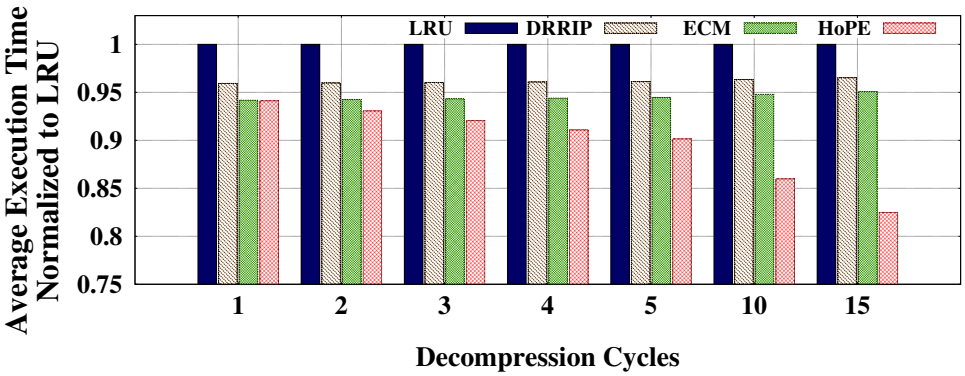


Figure 31: Performance sensitivity of HoPE to the decompression latency.

3.5 Conclusion

The performance gap between memory and CPU is still increasing, so determining how to use the fixed space allocated to on-chip caches effectively is one of the most important research challenges in the field of processor design. Memory compression can play an instrumental role in improving system performance and reducing energy consumption, because the post-compression reduction in data sizes can increase the logical effective capacity of a compressed memory system, without the need to increase the physical memory size. However, data compression techniques incur some undue overhead. The fundamental challenge is to avoid, or at least minimize, this overhead.

This work aims to mitigate one of the most important components of the overhead associated with data compression: the read-hit-decompression latency. The proposed Hot-cacheline Prediction for Early decompression (**HoPE**) framework consists of three fundamental techniques working in tandem: (1) *Hot-cacheline Prediction (HP)*, (2) *Early Decompression (ED)*, and (3) *Hit-history-Based Insertion (HBI)*. The first two policies work

collaboratively to identify the appropriate time that a compressed cacheline should be decompressed, in order to minimize the total execution time. The third policy takes advantage of the discovered correlation between compressibility attributes and sustained hit rates by predicting how often a cacheline will be hit.

To evaluate the effectiveness and efficiency of the proposed HoPE mechanism, we perform extensive simulations on memory traces obtained from multi-threaded benchmarks running on a full-system simulator. The simulations demonstrate that HoPE can reduce the read-hit decompression penalty in compressed LLCs by more than 60%, when compared to compressed caches using the LRU replacement policy, DRRIP [1], and the current state-of-the-art ECM mechanism [4]. This significant reduction in the read-hit penalty yields an average increase in overall system performance of 9.8% over LRU, 6.2% over DRRIP, and 4.6% over ECM, respectively.

CHAPTER 4

DUAL-PHASE COMPRESSION MECHANISM FOR HYBRID DRAM/PCM MAIN MEMORY ARCHITECTURES

The seemingly infinite abundance of on-chip computational resources puts an enormous strain on the memory sub-system of the CPU. More specifically, the desire to "feed the beast" necessitates both higher off-chip bandwidths and increased main memory capacities. Without these two pre-requisites, the vast amount of processing prowess afforded by the hardware will remain largely untapped. Although DRAM technology has been a mainstay of off-chip main memory implementations for the last few decades – because of its high read/write performance, large capacity, and economies of scale – it generally consumes more power than other memory technologies. While this issue has not hampered its pervasiveness in computer systems thus far, it could potentially transform into a critical show-stopper with the advent of multi/many-core processor architectures and emerging data-intensive applications. A further worrisome sign is the imminent saturation point in the scaling of DRAM technology [6]. Thus, the insatiable demand for additional memory capacity mandates the development of new architectural solutions that will enable the deployment of scalable memory systems.

Over the last few years, several new memory technologies have entered the fray trying to address some of the shortcomings of DRAM technology. One of the most promising new actors is Phase-Change Memory (PCM), which is gaining a foothold in the research community predominantly due to its ability to scale very deeply into the low nanometer regime [19]. PCM exploits the unique trait of chalcogenide glass to switch between two states – crystalline and amorphous – through the passage of an electric current. PCM offers the additional benefit of being non-volatile, which enables it to double as a storage medium as well. Its above-mentioned ability to scale down to extremely small feature sizes is envisaged as the key enabler to massive amounts of main memory at low cost. Moreover, PCM

consumes markedly less power than DRAM. However, there are some crippling limitations that prevent PCM from completely ousting DRAM from future systems: low write performance and limited long-term endurance. These drawbacks have led designers toward the adoption of various architectural techniques, which attempt to mitigate the deficiencies of PCM technology [20, 21, 22, 23, 24, 52, 53, 54, 55, 56, 57, 58, 59, 60].

The undisputed benefits and huge potential of PCM fully justify the significant effort expended in architecting feasible PCM-based implementations. One approach to compensating for the innate weaknesses of PCM technology is the introduction of various mechanisms throughout the memory sub-system of the CPU that actively interfere in the execution of memory operations [61, 62, 63, 64, 65]. Notable examples of such schemes include the reorganization of memory buffers [20], Flip-N-Write [21], and redundant-bit write removal and row shifting [22]. A higher-level memory system architectural approach employs a partial-write technique that flushes only a specific part of a dirty cacheline from the traditional cache [20]. Similarly, a segment-swapping mechanism can be used to periodically swap memory segments of high and low write accesses in order to achieve a wear-leveling effect (i.e., prolong the lifetime of PCM). Although these schemes reduce the actual number of write operations in PCM, they involve *multi-layered modifications* in the memory sub-system and a subsequent orchestration of activities at multiple levels of the hierarchy. In other words, these solutions involve widespread and invasive augmentations deep into the operation of the memory architecture. More importantly, these techniques tend to focus on only one of PCM's weaknesses: either performance, or prolongation of lifetime. Tackling *both* aspects simultaneously requires the combination of multiple mechanisms.

Another approach to circumventing the undesired characteristics of PCM is the adoption of hybridized memory structures, which utilize both DRAM and PCM memories, in order to combine the best of both worlds [23, 24, 25, 66, 67, 68, 69, 70]. Instead of using large capacities of power hungry DRAM memory, a hybrid main memory system uses large capacities of PCM, together with a small-capacity DRAM memory. The conventional

way to hybridize DRAM and PCM is by using the DRAM as an on/off-chip cache (i.e., an additional level in the memory hierarchy). This DRAM cache significantly reduces the number of PCM accesses, thus compensating for the low write performance of PCM and its limited endurance. However, the reaped benefits from such setup are heavily dependent on the application workloads and the configuration of the DRAM cache. Therefore, DRAM/PCM memory hybrids are not enough – by themselves – to adequately address all issues pertaining to the use of PCM. Substantial support from other layers in the memory sub-system is also pertinent.

Finally, memory compression has also been suggested for reducing the number of PCM writes [71]. However, this initial incarnation resorted to using compression only when the number of dirty words in a cacheline was greater than a certain threshold value [71]. Furthermore, the compression algorithm employed was not specifically optimized for the nuances of a PCM-based system (it was a generic compression algorithm). Consequently, the contribution of data compression was not found to be so pronounced. In addition to incurring compression/decompression overhead, the use of existing (i.e., conventional) data compression techniques for PCM may not be so effective, because of mismatches in the sizes of new and older compressed data segments. If the size of a new compressed item is larger, or smaller, than the size of the entity it replaces, then data must be moved around to accommodate the new item. Such data movement requires address space remapping and/or the generation of additional write traffic for compaction [72]. Clearly, the incorporation of efficient and effective data compression in PCM-based systems necessitates optimizations at both the algorithmic and architectural levels. The solution must be fully tuned to the underlying memory substrate, in order to seamlessly integrate with the operation of the system.

The aforementioned observations constitute the primary motivation of this work. The elemental driver of this chapter is two-fold: (a) the design of an efficient compression scheme that is highly optimized and tailored for PCM-based implementations, and (b) the

deployment of said scheme in an architecture that does not require any support from – or any modification to – the various layers of the cache/memory hierarchy. The ultimate goal is to design and evaluate a compression technique that is especially amenable to PCM-based environments and transparent to the operation of the remaining components of the memory sub-system; i.e., a self-contained and self-supported solution. Toward this end, we hereby propose a novel dual-phase compression technique that caters to the inherent limitations of PCM technology, and then explore its design-space over various architectural configurations. Without loss of generality, we choose to amalgamate our compression scheme with existing DRAM/PCM hybrid architectures, since the latter appear to be among the most viable and promising candidates for PCM integration into future CMPs. Note, however, that the idea presented here may also be used in PCM-only configurations with only minor modifications.

The main contributions of this work are:

- A tailored dual-phase compression mechanism for better performance and wear-leveling. The first phase optimizes a small-size DRAM cache by utilizing a simplified *word-level* compression algorithm, which considers the latency and effective capacity of the DRAM cache. The second phase adopts a *bit-level* compression algorithm for the large-size PCM itself, so as to further reduce the number of PCM accesses.

- Design-space exploration of the DRAM cache management, focusing on the DRAM cache size and the cache replacement policy, in order to maximize the reaped benefits.

- A multi-faceted wear-leveling technique that guarantees reasonable lifetime endurance for the PCM. This mechanism acts on the data after compression and intelligently balances wear-out within the PCM.

- The devised architecture can be cost-effectively implemented by only slightly modifying the memory controller. It does not require any architectural support from any

other entity in the CMP.

In order to validate the efficacy of the proposed architecture, we employ a simulation framework driven by traces extracted from real application workloads running on a full-system simulator. The dual-phase mechanism is thoroughly evaluated and demonstrated to yield 27.8% performance improvement and 30.5% energy reduction, on average, as compared to a baseline DRAM/PCM hybrid implementation. This improvement is further enhanced by up to 4.7% (in terms of performance) and 19.9% (in terms of energy), by exploring the design-space of the DRAM cache. Finally, the multi-faceted wear-leveling technique is shown to significantly prolong the lifetime of the PCM, up to 43.1×, with slight modifications required only at the memory-controller level.

The rest of the chapter is organized as follows: Section 4.1 serves as a preamble by providing a concise summary of memory devices and related prior work in this domain. Section 4.2 then proceeds with the description of the high-level concept of the proposed dual-phase compression scheme, while Sections 4.3 and 4.4 describe the design of the compressed DRAM cache architecture, and the PCM wear-leveling technique employed by the DPC mechanism, respectively. Section 4.5 presents the implementation details of the proposed design and analyzes the associated overhead. Next, Section 4.6 presents the employed evaluation framework, the various experiments, and accompanying analysis pertaining to the performance, energy consumption, and endurance of the proposed design. Finally, Section 4.7 concludes this chapter.

4.1 Background

4.1.1 Basics of Memory Devices and Systems

In order to accommodate the demands of increased memory bandwidth and capacity requirements, modern systems are expected to support massive amounts of off-chip main memory. Typically, memory chips are integrated together and installed as Dual In-line Memory Modules (DIMM). The memory chips on each DIMM are operated in parallel (or

interleaved) for high-bandwidth accesses. As a result, a DIMM's energy consumption is significant. For example, one 4 GB Double Data Rate 2 (DDR2) DIMM module consists of 36 DRAM chips (2 ranks, 18 chips per rank, and 8 banks per chip) and it consumes around 12.8 W in operating mode (interleaving), or 3.9 W in idle mode [43]. Since several DIMMs are, in fact, used in a single computer system, the proportion of total power consumption due to the main memory system is between 20 to 40% [73]. This percentage is predicted to sharply increase as we transition into the many-core era.

Recently, Phase-Change Memory has emerged as one of the most promising solutions for future memory systems due to the following attractive characteristics:

- Ability to scale down to diminutive feature sizes – PCM can scale down to under 10 nm, whereas DRAM's ability to scale even below 30 nm has been questioned [6, 74].
- Low power consumption – PCM has almost $1/3$ of DRAM's consumption when in operating mode, and almost zero consumption when in idle state. Moreover, unlike DRAM, no refresh is required.
- Non-volatility – No need to backup when the power is switched off.
- Fast read performance – PCM currently supports the DDR2 interface [75], which means that its bandwidth is up to par with that of DRAM.

At first glance, PCM technology appears to be an ideal non-volatile memory solution that can completely replace traditional memory devices, such as DRAM. However, there exist two main limitations that hinder its acceptance: low write performance and limited cell endurance. The PCM's write operation commences with the melting of the cell under hot temperature, followed by fast quenching (reset operation) or slow quenching (set operation), depending on the write value. This heating-and-cooling process generally takes more than 100 ns, and, consequently, write performance is bounded by this time, which is very difficult to reduce. Another issue with PCM technology is the limited endurance of its

cells. The repeated heating-and-cooling process tends to alter the cell characteristics, and the probability of malfunction increases with the number of writes to the same cell. After a certain number of write operations, the cell is worn out. In current implementations, the PCM cell's endurance is limited to 10^6–10^8 write cycles.

Even though these limitations restrict the practical use of PCM in real products, said technology's inherent advantages and potential are extremely attractive to system designers. Therefore, researchers are actively trying to mitigate PCM's deficiencies by developing appropriate *supporting* architectural solutions, which can aid and complement the operation of the PCM.

4.1.2 Compression Techniques in PCM

Compression techniques have generally been used to reduce the memory traffic and increase the effective capacity of memory systems. However, they require additional, non-negligible latency overhead during the compression/decompression processes. In addition, address remapping schemes – as well as somewhat complex compaction and/or reallocation processes – are mandatory, because of the variable size of the compressed data. Hence, compression techniques have not been widely adopted in real products. However, despite the non-negligible overhead, compression techniques constitute an attractive solution within the context of PCM, because the reduced data size after compression can simultaneously and significantly increase the performance (especially the write performance), lower the energy consumption, and improve endurance. Furthermore, if the remaining (saved) space is not used to store additional data, the overhead of address remapping and data compaction and/or reallocation can be eliminated. Fortunately, one can easily give up the capacity benefits of compression, because the main attraction of PCM is to provide more space than DRAM at the same cost. Hence, the remaining space (after compression) can instead be used to further enhance the lifetime of the PCM. We will demonstrate this effect later on in the chapter.

Compression techniques have been investigated for PCM-based main memory systems

in [71]. However, compression is only one of several alternatives explored in that paper, and it is only triggered when the other alternative schemes are not expected to work. Furthermore, the compression algorithm employed was not specifically optimized for the nuances of a PCM-based system. Consequently, the contribution of data compression was not found to be so important.

The authors of [72] indicate that conventional data compression techniques may not be so effective for PCM, because mismatches in the sizes of new and old compressed data segments may generate additional write traffic for compaction, and may require address space remapping. Therefore, the incorporation of efficient and effective data compression for PCM-based main memory systems necessitates optimizations at both the algorithmic and architectural levels; simply applying conventional compression techniques to PCM-based systems will not yield the desired effects.

Recently, encoding-based memory compression (utilizing a frequent pattern compression scheme [76]) was introduced as a way to reduce the number of PCM writes [77] and to efficiently exploit Multi-Level Cell (MLC) PCM [78]. However, the work in [77] does not address the issue of reducing the compression/decompression overhead. In addition, the encoding engine is embedded inside the PCM chip and requires modification of the PCM row-array structure. This implementation also requires extensive resource duplication, since all individual memory chips must have the same encoding tables and logic. Thus, the energy consumption of the PCM device increases. Moreover, the process of updating the frequent-value table is not trivial, and all the tag bits must be stored within the PCM device at a fixed position (i.e., the wear-leveling mechanism must also consider the position of the tag bits). Finally, the proposed scheme in [77] focuses only on enhancing the energy consumption and the endurance of the PCM.

On the other hand, the scheme proposed in this chapter can be effectively implemented within the memory controller alone, and does not require any architectural (or other) support from the system. These attributes render its potential incorporation into future CMPs

very straightforward. Most importantly, the technique proposed in this chapter simultaneously enhances the performance, energy consumption, and endurance (life-time), by applying a fine-tuned compression technique specifically tailored to the characteristics of PCM technology.

4.2 Dual-Phase Compression (DPC)

The principle of using data compression as a means to increase the effective capacity of caches and/or memory has been explored extensively by researchers. The compression algorithms employed in these conventional cache/memory systems suffer primarily from two afflictions: (1) they incur additional access delay, due to the compression/decompression overhead (this may be further pronounced in a PCM setting, due to PCM's low write performance); (2) they incur additional latency overhead for managing the variability in the size of compressed data, which requires extensive address remapping.

Our goal in this work is to overcome both of these limitations, so as to enable efficient and seamless data compression in DRAM/PCM hybrid architectures. The proposed Dual-Phase Compression (DPC) scheme for hybrid DRAM/PCM main memory systems effectively hides the additional compression/decompression-related latency while keeping a high compression ratio, by distributing it to two phases. Moreover, the DPC mechanism does not require any address remapping, thus eliminating this extra overhead.

Figure 32 illustrates the high-level view of the proposed DPC mechanism. The first

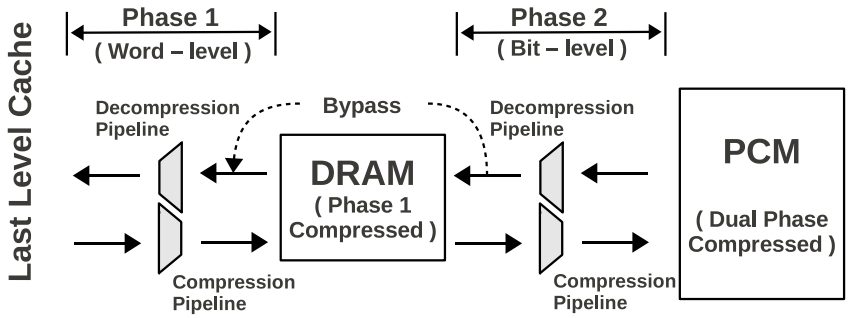


Figure 32: An overview of the proposed Dual-Phase Compression Mechanism, as applied to a hybrid DRAM/PCM main memory implementation.

phase of the compression process aims to not only minimize the additional compression/decompression related latency, but also to increase the effective DRAM cache capacity. By increasing the effective size of the DRAM cache, we reduce the DRAM cache misses and, therefore, the number of PCM accesses. The first phase compresses data at the *word-level* using a simple Successive Matching Algorithm (SMA). This approach allows for rapid access to critical words during memory accesses (*critical-word-first* scheme) without waiting until the decompression process is completely done. Consequently, the DRAM cache stores the compressed data of Phase 1 of the DPC scheme.

The second phase operates on the compressed data of the first phase to achieve even more data compaction; thus, the PCM stores data that has been further compressed. This phase compresses data at the *bit-level*, using a Frequent Pattern Algorithm (FPA). The DPC mechanism also incorporates a DRAM-bypass path that can be used to reduce the cache read-miss latency.

4.2.1 Phase 1: Word-level Compression with a Successive Matching Algorithm (SMA)

Figure 33 illustrates the SMA algorithm utilized in the first phase of the proposed DPC architecture. In this phase, a 64B data block (a cacheline containing 16 32-bit words) is compressed at the granularity of a single word (word-level compression). As the term "successive matching" suggests, compression is only performed between *adjacent* (consecutive) words in the 64B block. While compressing the original 64B data block, the *i* and *i+1* words are compared in the "Word Comparator Arrays" of Figure 33 for a possible match ($i = 0$ is the least significant word of the original 64B data block). The resulting compressed data block comprises of two parts: the *Compression Tag Area* and the *Compressed Data Area*. The Compression Tag Area consists of 16 bits, with each bit corresponding to one word in the original 64-B data block (remember, a 64B block holds 16 words). If two consecutive words are found to be identical, a '0' is written in the corresponding bit of the Compression Tag Area. Otherwise, a '1' is written. The first bit of the Compression Tag Area is always '1', since there is no *i-1* word to compare with. If the *i*-th tag bit is '1', then

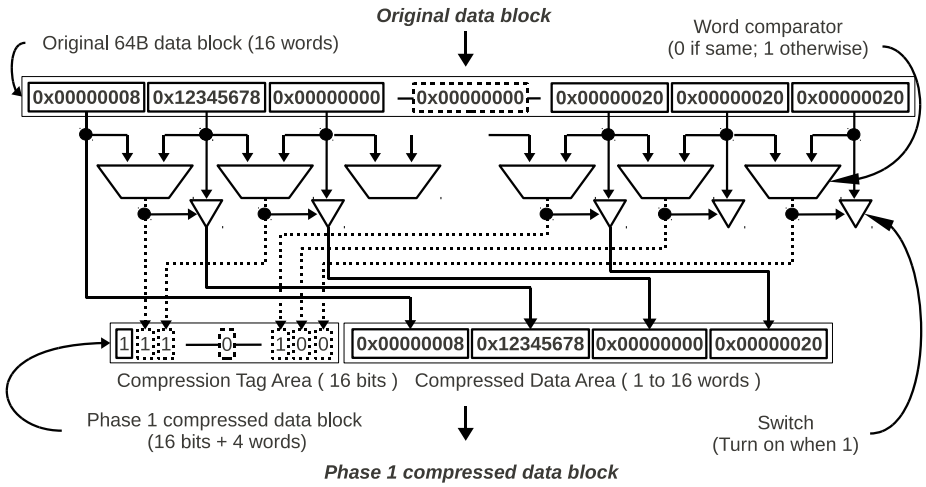


Figure 33: Phase 1 of the Dual-Phase Compression scheme employs a hardware-based Successive Matching Algorithm (SMA).

the *i+1* word is written from the original data block in the Compressed Data Area (Figure 33). Otherwise, nothing is written.

In summary, the Compression Tag Area is used to identify the following:

1. Which of the original 16 words have been compressed (all words with corresponding tag bit '0').

2. Whether the entire cacheline is compressed or not. If there are no '0' bits in the tag area, then the cacheline is uncompressed.

3. The size of the compressed data block. This is signified by the number of tag bits that have a value of '1'. For example, if four tag bits have a value of '1' (just like in Figure 33), then the Compressed Data Area consists of four 32-bit words.

Looking at the scenario presented in Figure 33, the first three tag bits are set to '1', because the first three words in the original data block are different. However, the 4^{th}–13^{th} tag bits are all set to '0', because the 3rd word in the original data block is repeated multiple times. Similarly, the last three words in the original data block are the same. Hence, the 14^{th} tag bit is set to '1', while the 15^{th} and 16^{th} bits are set to '0', to indicate compression.

In this example, the compression ratio is calculated as $\frac{16\ tag\ bits+(4\ words\times 32\ bits)}{16\ words\times 32\ bits} = 0.28$. Note that the closer the compression ratio is to zero, the better the compression efficiency.

The details of the hardware design issues related to this phase, including the design of the DRAM cache architecture, will be described in Section 4.3.

4.2.2 Phase 2: Bit-level Compression with a Frequent Pattern Algorithm (FPA)

Phase 2 of the proposed DPC architecture attempts to compact the compressed data of Phase 1 even more for storage into the PCM. In order to identify this additional redundancy that can be further compressed, the mechanism has to operate at a finer granularity than the word-level approach of Phase 1. Indeed, Phase 2 conducts compression at the *bit-level*.

One of the most efficient and fastest bit-level compression algorithms is FPA [79], which uses a frequent pattern encoding table, as illustrated in Figure 34. Note that this figure shows only the FPA-related data area, which is what is actually stored in the PCM. The table includes eight bit patterns that tend to appear frequently in real application workloads [80]. These eight patterns are known a priori and are converted to 3-bit prefix values (2^3=8) and a corresponding data value in their compressed form (see Figure 34). For example, a so called *'0'-run* (a pattern of all consecutive zeros in the word) corresponds to the first pattern in the table of Figure 34, which compresses to a prefix of '000'. In the example of the same figure, the third word of the compressed data from Phase-1 is compressed to the '000' prefix and a 1-bit data value of '0', because the original word is a '0'-run word. As shown, after the FPA procedure of Phase 2, the compressed data block from Phase 1 is further compressed to 76 bits. Compared to the original, pre-Phase-1 data (i.e., the initial uncompressed cacheline), the compression ratio is $\frac{76\ bits}{64\ bytes\times 8\ bits} = 0.15$. Compared to the post-Phase-1 compressed data, the Phase 2 compression ratio is $\frac{76\ bits}{144\ bits} = 0.52$.

The details of the hardware design issues related to this phase, including the tag area requirements for accessing the data area, and the wear-leveling techniques employed to protect the PCM, will be described in Section 4.4.

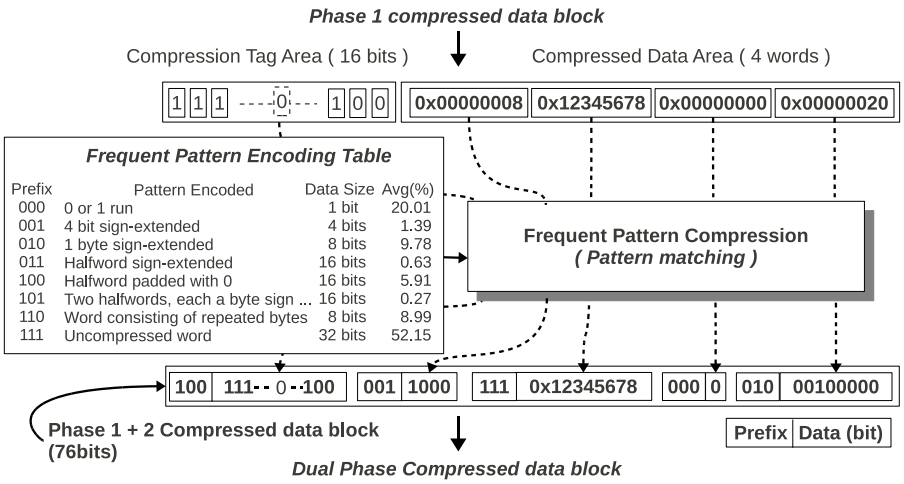


Figure 34: Phase 2 of the Dual-Phase Compression scheme employs a hardware-based Frequent Pattern Algorithm (FPA).

4.3 Compressed DRAM Cache Design

4.3.1 Structure of the Compressed DRAM Cache

In order to fully take advantage of the compression in Phase 1 of the DPC scheme, the DRAM cache must be able to accept more compressed cachelines than conventional uncompressed cachelines in a set. In other words, it must be able to accommodate cachelines of variable size without wasting available space due to placement restrictions. Such cache structures operate on the principle of decoupled, variable-segment caching [5]. The proposed cache architecture is depicted in Figure 35. While each set is *physically* 4-way set associative, we overlay a *logical* 16-way set associativity by using more tags and a segmented data area. More specifically, the data array of the cache is broken into 4B (single-word) segments, with 64 segments statically allocated to each cache set.

As previously described, the SMA algorithm compresses each 64B cacheline into anything between 1 (fully compressed) to 16 (uncompressed) words. Hence, each set in the proposed cache structure of Figure 35 can, technically, hold up to 64 compressed cachelines (in the extreme case where each cacheline is compressed to a 1 single 32-bit word, or segment). In order to support such high number of supported cachelines per set, the Tag Area size (see Figure 35) would have to increase dramatically. So, the critical question raised is

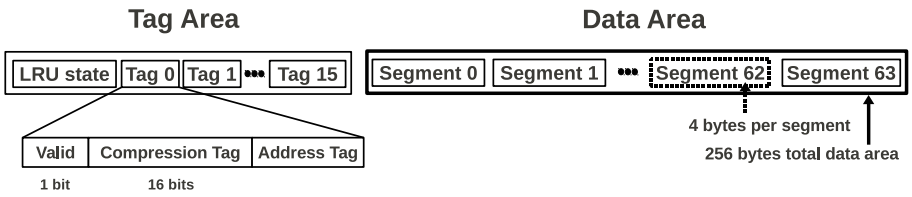


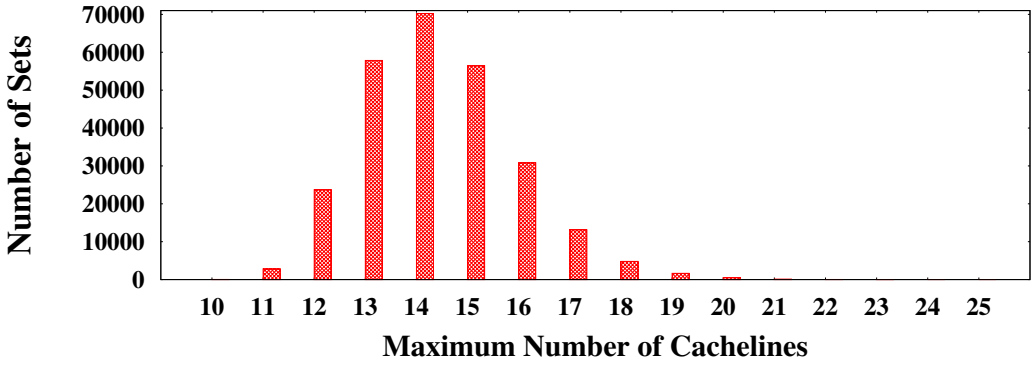
Figure 35: Illustration of a single set of the proposed DRAM cache structure, which is a similar implementation to a decoupled variable-segment DRAM cache [5].

whether real applications would benefit from such extreme capability (i.e., 64 compressed cachelines per set). In order to figure out the practical number of cachelines in each cache set, we track the maximum number of valid cachelines over all cache sets, as shown in the histogram of Figure 36(a), using memory traces from a full-system simulator. The specific histogram shown here as an example refers to the *bodytrack* benchmark. The details of the evaluation framework and the simulation parameters are described in Section 4.6. Figure 36(b) tracks the variation of the effective cache capacity (in terms of the number of valid cachelines) in two randomly chosen sets (A and B) over 50 billion CPU cycles, when running the same *bodytrack* benchmark. We only show these two sets here for illustration purposes. In fact, we conducted experiments tracking all the sets in all the benchmarks studied in Section 4.6, and the results indicate that real applications hardly exceed 16 valid cachelines per set, as illustrated in Figure 36(b). Thus, the proposed design can support up to 16 compressed lines per cache set, which is fully adequate for all the tested benchmark applications.

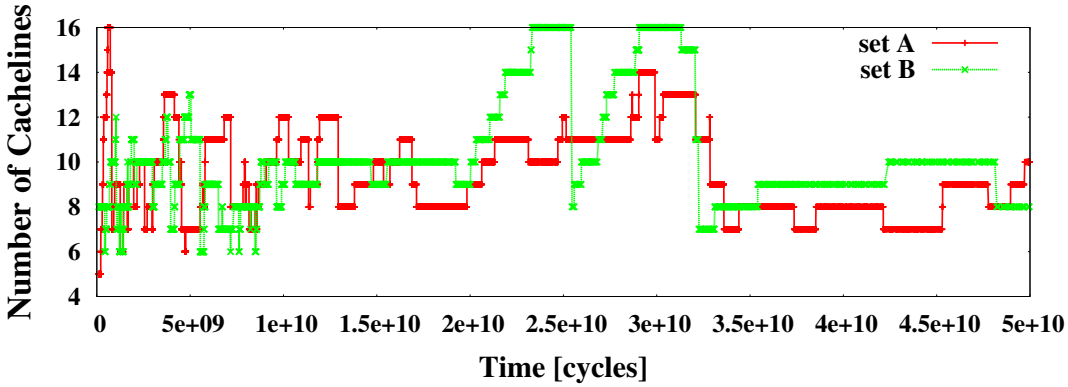
Overall, each cache set can store up to 16 compressed cachelines, or up to 4 uncompressed cachelines ($4\ lines \times 16\ words = 64\ words$). Therefore, each set can potentially increase its effective capacity by up to four times (when storing 16 compressed cachelines).

Each tag in the Tag Area (Figure 35) consists of a valid bit, the 16-bit Compression Tag (from Figure 33), and the address tag. The overhead, as opposed to a conventional cache, is the extra 16-bit Compression Tag, which is required for all 16 tags of each set.

Data segments are stored contiguously in Address Tag order. The offset for the first



(a) Histogram depicting the maximum number of valid cachelines over all cache sets.



(b) The variation in effective cache capacity (in terms of the number of valid cachelines) in two random sets.

Figure 36: Investigation of (a) the maximum number of valid cachelines over all cache sets, and (b) the variation in effective cache capacity (in terms of the number of valid cachelines) in two random sets when running the *bodytrack* benchmark (see Section 4.6 for the details of the evaluation framework and the simulation parameters). The number of valid cachelines in the two randomly selected cache sets (A and B) in Figure 36(b) is tracked over a period of 50 billion CPU cycles.

data segment of cacheline k (in a particular set) is

$$\text{segment\_offset}(k) = \sum_{i=0}^{k-1} \text{actual\_size}(i)$$

where $k \leq 15$ and actual_size(i) is the size of i^{th} compressed cacheline, in bytes, of the cacheline (whether compressed or not), given by

$$\text{actual\_size}(i) = \sum \text{the number of '1's in the valid compression tag}(i) \times 4 \text{ bytes}$$

where $0 \leq i \leq 15$. The 'segment_offset' and 'actual_size' parameters are used to target specific segments in the Data Area of the cache for address-tag matching.

To enable rapid access to a specific word in a cacheline, it is imperative for the technique to be able to swiftly locate the requested word in the compressed cacheline. This, in fact, is one of the potential drawbacks of conventional compression schemes: how does one locate a specific item within the compressed data (a) without having to decompress, and (b) without having to search the entire compressed block? The proposed Phase 1 using SMA avoids both of these complications by providing a *critical-word-first* capability: to access the *i*-th word from the original uncompressed data block, the controller simply counts the number of tag bits set to '1' in the 16-bit Compression Tag Area of the set (see Figure 33) from tag bit 0 to tag bit *i-1*. This number immediately tells the location of the requested i^{th} word in the compressed data block. This procedure pinpoints the required word *directly* with no need for decompression or searching. This is the reason why we distribute the compression process into two phases, so that only the first phase works before the compressed DRAM cache. If we do not distribute the compression process and use both SMA and FPA together before the compressed DRAM cache, there is always a non-negligible FPA decompression penalty whenever a compressed cacheline is a read-hit [5]. This is a valuable attribute that contributes to enormous performance benefits, as will be demonstrated in Section 4.6.

4.3.2 Compression- and PCM-aware DRAM Cache Design

The hit rate is the most important factor in traditional cache systems, and it is directly related to the size of the cache and how accurately the locality information of each cacheline is predicted when selecting a victim cacheline. In a conventional cache architecture, the size of the cache is physically fixed at design-time. Thus, only locality information is primarily exploited. However, it is known that the effective (logical) capacity in a compressed cache varies (a) over the duration of execution, and (b) with the achievable compression ratios of each application, as illustrated in Figure 36 (b). The effective capacity significantly affects the hit rate of the cache system [4]. Furthermore, the size of the evicted cacheline from the DRAM cache to the PCM also varies with the cache management policy, which also

significantly affects the PCM's performance. If an uncompressed cacheline is evicted from the DRAM cache, it requires more PCM writes than in the case of a compressed cacheline evictee.

These observations indicate that the effective size information of the DRAM cache and the actual size of the evicted data from the DRAM cache to the PCM should be considered together with the locality information when designing the DRAM cache. This holistic approach will help maximize the performance enhancement of the DPC mechanism.

The fundamental principle of the proposed DPC technique is to assist the cache architecture with suitable compression techniques, so that the effective capacity of the DRAM cache increases. This increase results in a reduction in the number of PCM accesses. The variation of the effective capacity for a single set is given by

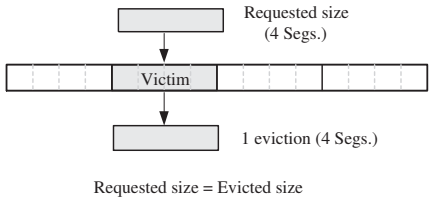
$$\text{physically 4-way} \leq \text{effective capacity} \leq \text{logically 16-way}$$

The minimum and maximum sizes of the capacity are bounded by the size of the physical data area and the tag size, respectively. Therefore, the number of cachelines that a single set can hold ranges from 4 to 16 (i.e., from a 4-way to a 16-way set), depending on the size of the compressed cachelines. For instance, if there are only uncompressed cachelines, the cache would operate with 4-way set associativity. Conversely, if most of the cachelines are highly compressible, then the cache would operate with 16-way set associativity.

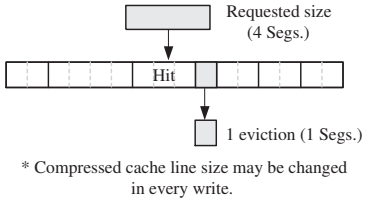
Currently, the Least-Recently Used (LRU) replacement policy is widely adopted, even in compressed caches. However, the LRU replacement policy may not be the best solution for maximizing the performance of the memory system in a compressed cache, where the effective capacity of the cache constantly varies. So, in this sub-section, we analyze the effect of exploiting the size information when designing the cache replacement policy.

There are two facts that directly affect the effective capacity of the cache: the size of the requested cacheline, and the size of the evicted cacheline. Figure 37 illustrates different eviction scenarios, based on the sizes of requested and evicted cachelines. Figure 37(a) shows the simple case of a conventional cache (i.e., without compression), where the sizes

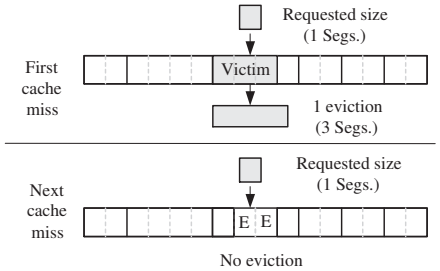
of the requested and evicted cachelines are always the same. In this case, no size complication ever arises in the cache management process. In the case of compressed caches, if the size of the requested cacheline is smaller than the available space in the set, no eviction is required, even in the case of a cache miss. However, if the size of the requested cacheline is bigger than the available space in the set, eviction is necessary, even in the case of a cache hit (specifically, a write hit), as shown in Figure 37(b). In a similar vein, if the size of a single evicted cacheline is large enough to hold more than two incoming cachelines, no eviction is necessary, even after a couple of consecutive cache misses, as shown in Figure 37(c). In this case, the effective capacity of the cache is temporarily increased. In the opposite case shown in Figure 37(d) – where the size of the originally selected victim cacheline is smaller than the size of the requested cacheline – more than one cacheline should be evicted, in order to accommodate one incoming cacheline. These observations indicate that the size information should be an important eviction metric when a compression mechanism is applied to the cache architecture.



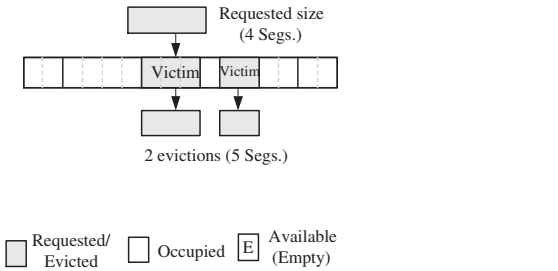
(a) Cache hits/misses in a conventional cache (i.e., with no compression), where the sizes of the requested and evicted cachelines are always the same.



(b) If the size of the requested cacheline is bigger than the available space in the set, eviction is necessary, even in the case of a cache hit (write hit).



(c) If the size of a single evicted cacheline is large enough to hold more than two incoming cachelines, no eviction is necessary, even after a second miss.



(d) If the size of the originally selected victim cacheline is smaller than the size of the requested cacheline, two evictions are necessary to accommodate one cache miss.

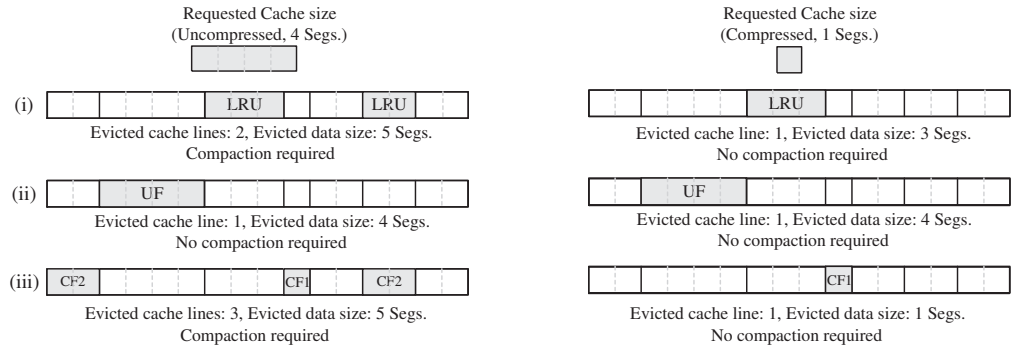
Figure 37: The size information of the requested and evicted cachelines is a very critical eviction metric in caches employing compression. Thus, the cache management policies should consider the size information during the eviction process.

Unlike the case of eviction (replacement), there is almost no way to control the size of the requested cacheline (except by changing the compression algorithm). Hence, our cache management and analysis mostly focuses on managing the size of the evicted cachelines, by devising a compression-aware DRAM cache replacement policy.

When selecting a victim cacheline – while simultaneously considering the sizes of the requested and evicted cachelines – three simple philosophies are possible: (1) uncompressed- (biggest) cache-line-first, (2) compressed- (biggest) cache-line-first, and (3) best-fit-cacheline-first. Note that we do not consider the best-fit solution here, because its implementation overhead is non-negligible and would result in a complicated hardware design. So, only the effects of the biggest-cacheline-first (uncompressed) and smallest-cacheline-first (compressed) replacement policies will be evaluated later on.

Figure 38 qualitatively compares the effects of the DRAM cache replacement policy, in terms of the sizes of the requested and evicted cace line sizes, and the effective capacity of the DRAM cache itself. Note that the evicted line is sent to the PCM. Figure 38(a) presents an example where the requested cacheline size is large (i.e., uncompressed cacheline), whereas Figure 38(b) shows an example where the requested cacheline size is small (i.e., compressed cacheline). In both examples, each uncompressed cacheline occupies 4 segments, while fully compressed cachelines occupy only 1 segment. Each of the two examples investigates three different replacement policies, presented from top to bottom in the figure: (i) LRU, (ii) biggest-cacheline-first (Uncompressed-First, UF), and (iii) smallest-cacheline-first (Compressed-First, CF). Initially, the example cache set includes 7 cachelines for all three replacement policies.

Starting with the big-size incoming request example – Figure 38(a) – the requested cacheline size is 4 segments. Under the LRU replacement policy of Figure 38(a)(i), if the size of the LRU-designated cacheline is 3 segments, at least two cachelines (LRU and the second-ranked LRU) must be evicted. As a result, the number of cachelines in the set decreases to 6. Furthermore, compaction is necessary to provide contiguous space for the



(a) An example where the requested cacheline size is large (i.e., uncompressed cacheline).

(b) An example where the requested cacheline size is small (i.e., compressed cacheline).

Figure 38: The effects of the DRAM cache replacement policy, in terms of the sizes of the requested and evicted cace line sizes, and the effective capacity of the DRAM cache itself. Three different replacement policies are investigated in both examples: (i) LRU, (ii) biggest-cacheline-first (Uncompressed-First, UF), and (iii) smallest-cacheline-first (Compressed-First, CF).

requested cacheline. In a similar manner, in the case of the compressed-first replacement policy of Figure 38(a)(iii), 3 cachelines must be evicted (the three smallest-sized lines, marked as CF1 and CF2) to make enough room for the new line, and a compaction operation is also needed. As a result, the number of cachelines remaining in the set is 5. This overhead (eviction of more than one cachelines, which degrades the effective capacity of the DRAM cache, and compaction) significantly impacts the performance of the memory system. On the other hand, under the uncompressed-first replacement policy of Figure 38(a)(ii), only one cacheline eviction is necessary (line marked as UF), without any compaction. Moreover, the number of cachelines in the set remains the same (i.e., 7).

In the small-size incoming request example – Figure 38(b) – the requested cacheline size is 1 segment. Under all three replacement policies (i)–(iii), only one cacheline is evicted, without any compaction overhead. Except in the case of the compressed-first replacement policy of Figure 38(b)(iii), the evicted data size is larger than the requested cacheline size. Large evicted data sizes tend to negatively affect the PCM, which has a slow write latency. At the same time, however, large evicted data sizes positively affect the DRAM cache, because the extra space remaining after the replacement of the large evicted

cacheline can be effectively used to hold more cachelines, thus increasing the effective capacity of the set. Additionally, the extra incoming cachelines of subsequent cache misses may not even cause further evictions, since the space is already available.

Intuitively, evicting an uncompressed cacheline first would maximize the effective capacity of the DRAM cache – because one eviction could accommodate at least one requested cacheline – and it would minimize the number of evicted cachelines. Indeed, our experiments (see Section 4.6.4) indicate that the uncompressed-cacheline-first policy maximally increases the effective capacity. However, as mentioned above, the eviction of uncompressed cachelines to the PCM imposes non-negligible overhead, in terms of performance and energy consumption, due to the PCM's slow write latency. As corroborated by simulations, writing large-size data to the PCM severely delays the application's execution time – even in the presence of a write buffer – and significantly increases the PCM's energy consumption. On the other hand, the compressed-cacheline-first policy (i.e., evict a compressed cacheline first) negatively affects the DRAM cache's effective capacity, but it positively impacts the PCM, by minimizing the number of PCM write operations. Based on these two simple observations, it is clear that considering the information of the evicted data size is helpful in maximizing the effective DRAM cache capacity, but it could be detrimental to the performance of the PCM. This implies that the consideration of only the size information has contradictory effects on the DRAM cache and the PCM.

To further complicate matters, in addition to the size information of the evicted cachelines, we still have to consider the cache's behavior with respect to *locality*, because the thrashing of large-size (uncompressed) cachelines with high locality and the prolonged presence in the cache of small-size cachelines with low locality are certainly undesirable artifacts. Hence, the size of the evicted data size from the DRAM cache to the PCM *and* locality information should be considered simultaneously and with a balanced approach. Obviously, the presence of so many competing and often adversarial design parameters makes it impossible to derive a single memory management policy that satisfies all constraints

simultaneously. Instead, the final design decision should be made based on a pre-defined desired tradeoff. In an effort to aid the designer in assessing all possibilities, this work will explore the design-space of the DRAM cache replacement policies by concurrently considering all salient properties.

The goal is to strike an efficient balance between the effective DRAM cache capacity and the PCM's write delay, based on a *two-level priority mechanism*. This mechanism will guide the DRAM cache replacement policy by first choosing a victim pool based on a *primary criterion* (first priority level). The actual victim will then be selected based on a *secondary criterion* (second priority level). Three fundamental cacheline criteria will be considered in this work. The first is based on the cacheline's size, i.e., uncompressed/compressed. The second criterion pertains to the line's modification status and aims to select the *clean* cachelines. This criterion tries to minimize the PCM's write delay, since clean lines evicted from the DRAM cache do not have to be written to the PCM. In addition to these two criteria, the locality behavior is also considered as a third tie-breaking criterion. This is achieved through management of the LRU bits. For example, if more than one replacement candidates are found in a cache set after the two-level priority filtering, the least-recently used one will be selected as the victim among the candidates.

Hence, assuming that we assign the first-level eviction priority to the uncompressed cachelines, the effective capacity of the DRAM cache will increase at the expense of the PCM's performance. To mitigate the latter, we can assign the second-level priority to the clean cachelines (among the selected uncompressed cachelines), so as not to impose excessive PCM write operations. It will be demonstrated that even though this approach slightly degrades the hit rate of the DRAM cache – as compared to the blind uncompressed-cacheline-first policy – it strikes an effective tradeoff between minimizing the PCM overhead and reaping the benefits of the uncompressed-cacheline-first policy. The net effect is a notable enhancement in the overall performance of the memory system. This type of configuration may be suitable for applications whose memory access frequency is high,

but mostly for read operations, i.e., where clean misses happen more frequently than dirty misses.

Conversely, if we assign the first-level eviction priority to the compressed cachelines, and the second-level priority to the clean cachelines, the PCM's performance will benefit at the expense of the DRAM cache's effective capacity. Such configuration will work for applications where write-dominant memory accesses are observed.

Finally, we may also reverse the priority criteria and assign the first-level priority to the clean cachelines, and the second-level priority to the uncompressed or compressed cachelines. By changing the priority order and trying different combinations, one may explore the impact of the DRAM cache replacement policies on the overall performance of the memory system. The detailed design-space exploration of the DRAM cache management policies described in this sub-section and all associated simulation experiments are presented in Section 4.6.

4.4 A Multi-faceted Wear-leveling Technique for Compressed PCM

One of the critical limitations of PCM devices, which impedes their widespread adoption in real-world memory systems, is limited endurance. A PCM device is expected to withstand only 10^6 to 10^8 write cycles – by current technology standards – before it fails. Without any additional supporting techniques, this amounts to only a few *days* worth of use under memory-intensive applications. Although hybridized DRAM/PCM architectures significantly reduce the number of write accesses to the PCM, it is still not an adequate solution – by itself – to prolong the PCM's lifetime to reasonable ranges. Several researchers have proposed additional *wear-leveling techniques*, which require some support from various levels in the memory hierarchy. In this section, we demonstrate that the proposed DPC architecture also works well with PCM wear-leveling mechanisms. For this purpose, a multi-faceted wear-leveling technique supported by the proposed DPC scheme is introduced. The differentiating trait of our approach is the fact that it does not require any

support from other levels of the memory hierarchy. In fact, the only support required is from the memory controller, with minimal timing and area overheads.

In general, wear-leveling techniques count the number of writes and swap the memory space if the difference between the maximum write count and the minimum write count is greater than a certain threshold. Of course, in this case, address remapping techniques, which incur additional timing and area overhead, are also required. The effect and corresponding overhead of wear-leveling techniques vary depending on the fundamental unit (or granularity) of wear-leveling. A finer-grained global wear-leveling technique can yield better results, but it will also incur a larger overhead. This implies a tradeoff decision between efficiency and cost.

In this work, we propose a multi-faceted wear-leveling technique for enhancing the wear-leveling efficiency with minimal overhead. We also try to avoid any severe remapping overhead associated with memory space swapping. Figure 39 illustrates a high-level overview of our compression-based, multi-faceted wear-leveling technique. We logically classify the PCM memory space into three granularity units: a *page* (logically the same unit as an operating system page; 4 KB in our configuration), a *line* (logically the same as a DRAM cacheline; 64 B), and a *segment* (a minimum memory operation unit: one word; 4 B). One page consists of 64 lines, and each line consists of 16 segments. Note that these unit-size values can be easily adjusted without any severe modifications. Based on these three types of granularity units, we devise two wear-leveling techniques; a Compression-based Segment Rotation (CSR) for local wear-leveling within a line, and a small-size Local-Count- (LC)-based OS-level page swapping for global wear-leveling between the pages. The details of these schemes will be analyzed in the following two sub-sections.

4.4.1 A Compressed-data-based Segment Rotation (CSR) Scheme for Intra-line Wear-leveling

The minimum unit of flushed-out data from the DRAM cache is a cacheline – normally 64 B – in this work. However, after applying our DPC mechanism, the effective size of a

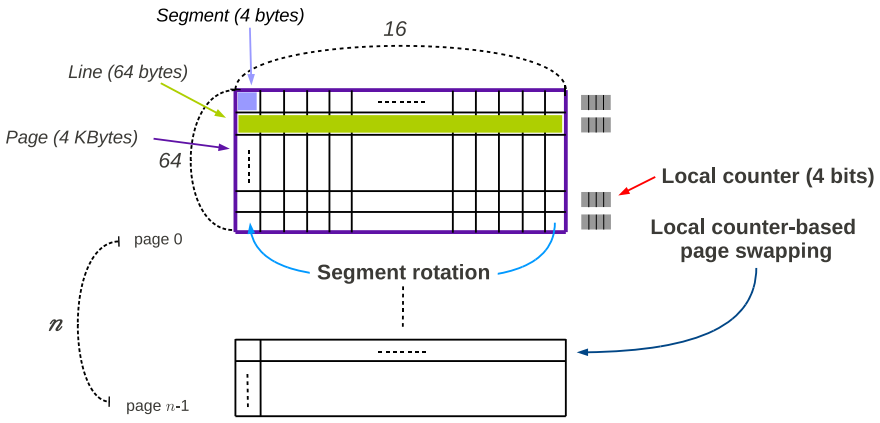


Figure 39: A high-level overview of the proposed multi-faceted wear-leveling technique.

cacheline is reduced to less than half of the original, on average. In most general-purpose compression algorithms, the remaining space is used for storing more data, in order to enhance the effective capacity of the memory system. Instead, our CSR scheme uses this remaining space to enhance the endurance of PCM under the assumption that the PCM device already provides enough memory space. This is not unrealistic, since PCM is an extremely scalable memory architecture offering massive storage density.

Figure 40 depicts the basic principle of the proposed CSR technique, which is only performed at the line level. Note that each cacheline has an associated hardware-based Local Counter (LC), which keeps track of the writes to the line. Without loss of generality, the size of each LC is chosen to be 4 bits, which will be shown later on to yield significant improvements in the PCM's lifetime. The purpose of this LC is to propagate write-count information to the page-level wear-leveling technique (see next sub-section) with small counting overhead. Note that the LC value is not incremented on every single write. Instead, the whole process unfolds as follows: the first write operation in Figure 40 only uses the first 7 segments of the available line space, because the compressed size of the flushed-out cacheline happens to be 7 words long (i.e., 28 B). At the second write, the data is stored in the following segments, from the 8^{th} to the 13^{th} (occupying 6 segments in the line). If the data spills out of the last segment of the line, the remaining information wraps around

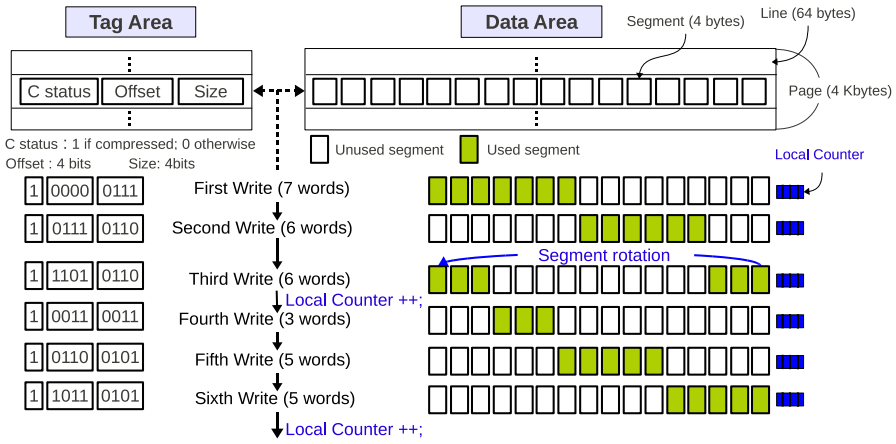


Figure 40: The fundamental operating principle of the proposed Compression-based Segment Rotation (CSR) process for local wear-leveling within a line. This wear-leveling technique takes place within the PCM device.

to the first segments of the line, as shown in the case of the third write in Figure 40. It is precisely at this point (i.e., whenever a wrap-around occurs, or whenever a write reaches the last segment of the line) that the LC value of the specific line is incremented. This rotation process within each line ensures that all segments in a line are evenly worn-out.

The 9 extra bits in the Tag Area (left-hand side of Figure 40) are used to indicate the compression status (1 bit), the offset in segments (4 bits), i.e., the start of the cacheline, and the size in segments of the compressed cacheline (4 bits).

The gist of our technique is the intra-line wear-leveling process achieved by CSR. Of course, CSR's benefit relies on the compression rates of the workloads, which are not uniformly distributed. Regardless, intra-line segment rotation always ensures uniform wear-out within each line.

4.4.2 A Local Counter-based Page swapping (PS) Technique for Global Wear-leveling

Since our CSR-based wear-leveling technique only operates within a cacheline, we also devise a global wear-leveling technique. The most commonly used global wear-leveling process is page swapping. When designing a page swapping mechanism, the most important factor is an accurate decision as to which pages should be swapped and when they

should be swapped. In order to make an accurate decision, the information about the number of writes already performed in each page should be accurate. If we just use one counter per page – for the sake of reducing the cost of counters – then we may fall victim to an over-counting problem. For example, let us assume that there are two pattern types of 10 consecutive writes to the same page. One pattern type writes the data to the exact same address – i.e., the same page and the same offset within the page – for 10 times. The other pattern type writes the data to the same page, but with different offsets, for 10 times. In the absence of a specific scheme for distinguishing the page offset within a page, both cases will increase the page-level counter by 10, even though the real increase in the write count in the latter case should be just 1, rather than 10. This over-counting problem may result in inaccurate swapping decisions. In order to ensure accurate counts, the ideal method is to keep one counter per word, but this is prohibitively expensive, due to the extremely large area overhead.

To solve this type of over-counting problem at a reasonable cost, we use the LCs described in the previous sub-section. Specifically, one 4-bit LC per single cacheline is employed, together with a page-level global counter. This setup is inspired by the work in [72]. Since the basic unit of operation of the DRAM/PCM hybrid main memory is the DRAM's cacheline, and our CSR technique already distributes the write requests evenly to all segments within a line, one LC per line does not decrease the accuracy of counting. Remember that the LC is increased only when there is a wrap-around write operation in the line. If any of the LCs in the same page overflows, the memory controller increases its corresponding page-level global counter by 1, and resets all the LCs in a page to zero. Simulation results indicate that the use of these small-size LCs mitigates the over-counting problem significantly.

Once we have the accurate counting scheme with a reasonable overhead, the page swapping part can easily be done by either using (hardware-based) memory controllers [22], or

the operating system [25, 81]. In general, hardware-controlled page swapping methods exhibit better efficiency, but require non-negligible additional page remapping overhead. So, in this work, we choose to use a similar OS-controlled page swapping methodology. Since most operating systems use page-mapping tables for the virtual memory architecture, there is no additional mapping overhead for page swapping. When a page is newly allocated by the OS, the OS itself can select a new physical page that has the minimum write count from the free-page list. If the OS detects that the difference between the maximum and minimum counts is greater than a predefined threshold, the page swapping operation is initiated. Although the page swapping operation incurs non-negligible overhead, it is known that the observed overhead is minimal, because swap operations do not happen frequently enough to become a major issue [25]. Hence, we ignore the performance overhead of this page swapping operation.

4.5 Implementation

The compression/decompression itself is not our contribution. Instead, we tailor the traditional compression algorithm to the nuances of DRAM/PCM hybrids. Figure 41 presents the memory system hierarchy and accompanying memory controller architecture, which consists of a DRAM cache controller, two compression/decompression engines, DRAM and PCM device controllers, a tag memory space, and (potentially embedded) DRAM. In DRAM/PCM hybrid memory implementations, the DRAM serves as a cache for the PCM (i.e., it forms an extra level in the memory hierarchy). Unlike the existing approaches that embed the compression/decompression engines inside of the memory device itself, all components in the proposed architecture – including compression/decompression engines and tag memory – reside in the memory controller and (e)DRAM. This results in a three-fold advantage: (a) Resource duplication is avoided (the individual memory devices do not need to have any compression-related logic); (b) One may easily change the compression algorithm, or any other configuration (such as the FPV table, which is also implemented

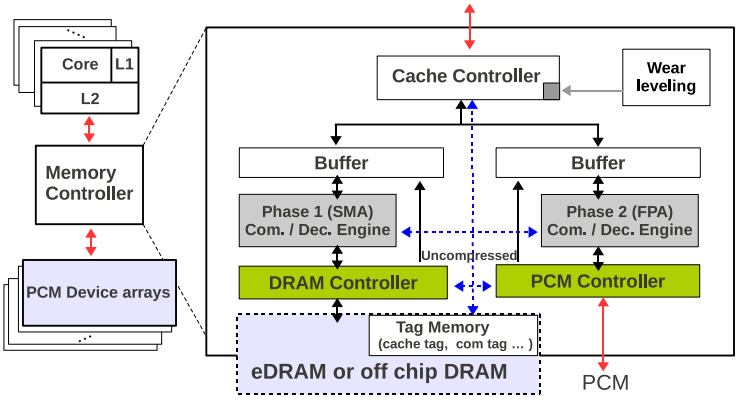


Figure 41: Implementation of the DPC Mechanism, as applied to a hybrid DRAM/PCM main memory implementation. In said implementations, the (e)DRAM serves as an on-chip cache for the substantially larger PCM device.

in the memory controller); (c) There is no special requirement for the management of the tag memory. If the tag bits are stored within the PCM device itself (like in previous approaches), there is additional access delay in the PCM, and the fixed location (addresses) of the tag bits significantly increases the complexity of the wear-level control mechanism. On the contrary, the proposed mechanism stores the tag bits within the (e)DRAM. Finally, the new DPC technique is transparent to the higher cache levels and to the PCM device architecture, so no other components need to be modified to adopt the new mechanism. Depending on the implemented technology and required DRAM capacity, the DRAM can also be integrated on-chip, as embedded DRAM. In such a scenario, the entirety of the proposed DPC architecture can be embedded within the memory controller. Hence, the DPC scheme can be efficiently integrated within any conventional memory system, without requiring any architectural changes.

Note that, in the proposed DPC scheme, memory addressing is not affected, as opposed to other DRAM/PCM hybrid techniques, even though we employ compression. In fact, this is the most important benefit of this new technique. Regarding Phase 1 of DPC, compression does not involve any address remapping: the remaining cache line space (the space gained after compression) is simply used to store additional cache lines. This means that the system can store more cache lines, depending on the compression ratio achieved. This

scheme is similar in principle to [5], but a different compression algorithm is employed. In Phase 2, the additional gained space (after the second compression) within the PCM line is not used to store additional data (i.e., no compaction is performed), which would otherwise be stored in a different address space. This is the reason why no address remapping is required in Phase 2. Of course, the offset within a line can vary, but this is easily managed by the simple hardware embedded in the memory controller.

4.5.1 Overhead Analysis

The DPC mechanism presented here requires the implementation of two compression engines (Phase 1 and Phase 2) and a tag memory, and to modify the DRAM cache controller to support the new scheme. The new and/or modified components required by the proposed architecture are shown as darkened boxes in Figure 41. These additional (modified) components may affect both the implementation cost and the access time. This sub-section analyzes the implementation overhead in terms of storage, hardware (control logic) cost, and impact on access time.

4.5.1.1 Storage Overhead

Since the DPC mechanism manages all the extra information – such as extra tags, offsets, and the size information of the compressed data in each phase – additional storage space must be provided. In Phase 1, the compressed DRAM cache requires 16 extra compression tag bits and additional tag space for 12 extra sets (to provide the extra associativity per cache set). This storage overhead, in terms of extra bits, only amounts to approximately 3% of the total capacity of the DRAM cache. However, if there are at least two identical consecutive words in the original 16-word data block, then this overhead is fully amortized and yields immediate compaction benefits. In fact, we observed that a significant portion of the cache lines of real application workloads contain several identical consecutive words, which allows the proposed compression scheme to reap enormous benefits (see Figure 42 in Section 4.6).

In Phase 2, nine bits of additional tag memory space per line are needed (i.e., a 1.8% space overhead) for indicating the compression status (C status – 1 bit), the offset (4 bits), and the size of valid compressed data (4 bits); see Figure 40.

Additionally, for the LC-based page swapping technique (Section 4.4.2), we use merely 0.9% of additional memory space, as computed by $\frac{32\text{–}bit\ global\ counter\ +\ (4\text{–}bit\ local\ counter \times 64\ lines)}{4\ KB\ (size\ of\ an\ OS\ page)}$.

4.5.1.2 Hardware Overhead of the Control Logic

The main overhead of the control logic stems from the two compression/decompression engines. Since the compression algorithm used in Phase 1 is very simple, its implementation cost is almost negligible compared to the control logic of the remaining memory system. Only one set of 32-bit comparators and switches is needed, one 6-bit parallel-prefix adder, and minor glue logic, because a cache-line of data (64 B) arrives from the L2 cache word-by-word, over different clock cycles. For better pipelining, we may use more than one set of comparators and switches, but we definitely do not need 16 in the memory controller. In this work, we compare each word serially in a 64 B cache line. We measure the gate count for these components using the method proposed in [82]. The number of required gates is 394 for compression/decompression. This amounts to only 0.81% of a commercial DDR2 SDRAM memory controller Intellectual Property (IP) block (48,500 gates). This means that the additional power consumption due to this compression/decompression logic is almost negligible compared to the relatively heavy power consumption in the memory-sub system (generally, on the order of several Watts). So we ignore this additional energy consumption in our evaluation.

The implementation cost of the FPA used in Phase 2 is already analyzed in [5, 80]. The area overhead is 0.184mm^2 at 45 nm technology, and power consumption is merely 0.273 W. We include this additional power consumption in our evaluation.

4.5.1.3 Access Time Analysis

The DPC mechanism performs compression and decompression when there is a request from the higher levels of the cache hierarchy. Thus, additional latency for compression/decompression and/or reading/writing the tag memory are unavoidable. This additional latency is non-negligible in existing compression-based memory systems. However, the proposed DPC architecture distributes this latency overhead over two phases, whereby the latency in Phase 1 is shortened, while the latency in Phase 2 is hidden by the relative long latency of the PCM. As a result, the overall latency is reduced.

We first analyze the compression/decompression delay of the SMA algorithm used in Phase 1, by calculating the critical path delay of the required logic, based on FO4 delays. The critical path is measured as 14.5 FO4 delays. At the 45 nm technology node, one FO4 delay translates to about 15 ps, so a critical path of 14.5 FO4 delays is less than 1 memory clock cycle in a DDR2-200 memory setup. Therefore, we add an overhead of 1 memory clock cycle per compression/decompression operation in our evaluations.

The FPA algorithm used in Phase 2 imposes a 5-cycle delay for decompression, and 1-cycle delay for compression [79]. These values are higher than those in Phase 1. However, the Phase 2 operations are not so latency-sensitive, because most memory operations are filtered by the DRAM cache and, hence, they do not require Phase 2 compression for PCM storage. Moreover, the 5-cycle Phase 2 compression/decompression delays are almost negligible compared to the PCM access latency. Regardless, we include these delays in all evaluation experiments.

Regarding the tag memory access overhead, there is almost no additional latency in Phase 1, because the compression tag bits are accessed simultaneously with the normal cache tag bits. In fact, this latency may even decrease, because of the critical-word-first scheme exploited in this phase. This benefit will be analyzed in the next section. For Phase 2 operations, the extra 9-bit tag data devoted to the DPC mechanism (see Figure 40) should be read and updated, but this data is located within the (e)DRAM cache, so the 9-bit

information can be accessed as part of the tag area access of the DRAM cache. Hence, there is no additional timing overhead in accessing this data.

4.6 Experimental Evaluation

4.6.1 Simulation Framework

An internally-developed trace-driven simulator is employed to evaluate the proposed DPC architecture. The traces have been extracted from the Simics full-system simulator [40] with timing information in CPU cycles. The simulated system is an 8-core in-order processor with shared Last-Level Cache (LLC). All the details of the simulation configuration are described in Table 5.

We investigate a total of four memory system design configurations: (1) DRAM-only, (2) PCM-only, (3) DRAM/PCM hybrid, and (4) DRAM/PCM hybrid with the new DPC technique. The DRAM-only and PCM-only memory configurations use 8 GB DRAM (large DRAM) modules and 8 GB PCM modules, respectively, while a 64 MB DRAM (small DRAM) and 8 GB PCM module are used for the DRAM/PCM hybrid memory configuration. Note that the DRAM-only configuration serves as an ideal reference point, in terms of performance (since it only uses DRAM technology, which offers the best performance). It is included here to enable a meaningful comparison between the DRAM/PCM hybrids.

Based on design-space exploration with realistic workloads, the improvements in performance and energy consumption obtained by increasing the DRAM cache size of the

Table 5: Simulated system parameters for evaluating DPC.

Number of CMP Cores	8
Processor Core Type	UltraSPARC-III+, 2 GHz
L1 caches (Private)	I- and D-cache: 64 KB, 4-way 64 B block
L2 caches (Shared)	1 MB, 4-way 64 B block
DRAM memory (large)	DDR2 8 GB
DRAM memory (small)	DDR2 64 MB

DRAM/PCM hybrid configurations tend to saturate at around 256 MB for most applications. Despite this observation, we set the size of the DRAM cache to 64 MB (for the DRAM/PCM hybrid memory configurations), because the proposed DPC technique increases the effective capacity of the DRAM cache by 2 to 3 times, on average. Note that this smaller size of DRAM cache will generate more stress toward the PCM, which will make our final estimated lifetime of the PCM device worse.

Since our focus is on improving performance and energy consumption within hybrid DRAM/PCM environments, we choose not to use a larger size of PCM, which would experience more performance benefits due to fewer page faults. Such a decision is deemed orthogonal to this work; should a designer decide to use a larger PCM, the results will be even better. The cacheline size is set to 64 B, which is a common configuration in most systems. Table 6 shows the performance and energy parameters used in our evaluations, which are derived from state-of-the-art academic and industrial references. Since our focus is to explore the relative effect of compression in terms of performance, power, and lifetime, we consider a constant memory device delay. Note that PCM cannot support burst-writes, due to heavy power consumption in the charge pump. Thus, most PCM prototypes only support a Synchronous Double Data Rate (SDDR) interface for read operations. This implies that reducing the data size by even just a few words can significantly enhance performance and energy consumption.

Table 6: The energy and performance models used for the PCM and DRAM devices.

Parameter		PCM	Large DRAM	Small DRAM
Latency(ns)	Read	75	30	30
	Write	200	30	40
Energy(nJ)[1]	Read	47.02	449.10	244.98
	Write	2,504.35	431.22	227.10
	Idle (W)	0	19.45	2.45

[1] Energy consumption per 64 B (i.e., 1 cacheline) access

Table 7: Memory access characteristics of the benchmark applications used (PARSEC benchmark suite [2]).

Name	Application	Description	R/W ratio	Mem. acc./CPU-cycle
Group 1	freqmine	Frequent itemset mining	1.55	0.01630
	raytrace	Real-time raytracing	27.54	0.00027
Group 2	blackscholes	Option pricing with blackscholes PDE	5.96	0.00023
	bodytrack	Body tracking of a person	5.96	0.00083
Group 3	streamcluster	Online clustering of an input stream	8.24	0.00015
	x264	H.264 video encoding	1.78	0.00232
Group 4	dedup	Compression with data deduplications	1.39	0.00474
	vips	Image processing	1.32	0.00164
Group 5	ferret	Content similarity search server	1.52	0.00091
	facesim	Simulates the motions of a human face	6.58	0.00022
Group 6	fluidanimate	Fluiddynamics for animation purposes	1.39	0.00474
	swaptions	Pricing of a portfolio of swaptions	11.11	0.00058

We selected several benchmarks from the PARSEC benchmark suite [2]. Their memory-access characteristics – such as the ratio of read operations normalized to the write operations, and the frequency of main memory accesses – are shown in Table 7. Based on the achievable compression ratios in each application (shown in Figure 42(a), whose details are explained in the next sub-section), we categorized the 12 benchmarks of Table 7 into six groups (Groups 1 to 6). Group 1 consists of the two most compressible applications, while Group 6 consists of the two least compressible applications. Groups 2 to 5 lie in-between Groups 1 and 6, in terms of achievable compression ratios.

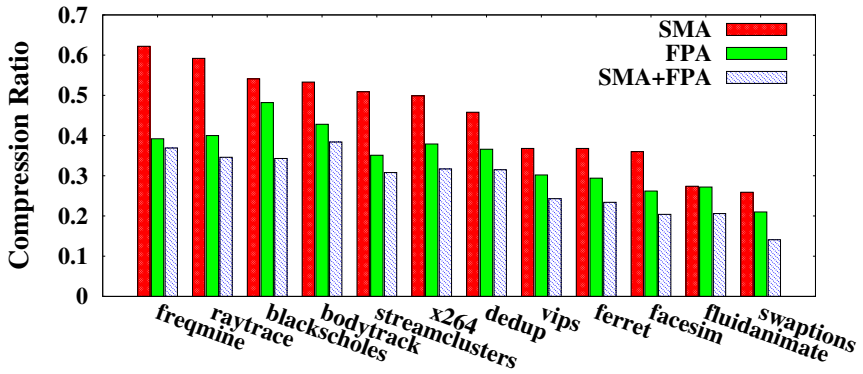
4.6.2 Data Compression Efficiency

Although the compression algorithm itself is not our contribution, we analyze the efficiency of three compression schemes: SMA-only, FPA-only, and the proposed DPC (SMA+FPA). For the FPA scheme, we use the same eight frequent patterns used in [79], both for FPA-only and SMA+FPA, because we found no significant pattern changes after the SMA compression in Phase 1 of the DPC scheme. We analyze the compression efficiency in terms of the average compression ratio achieved and the breakdown of cacheline sizes after compression.

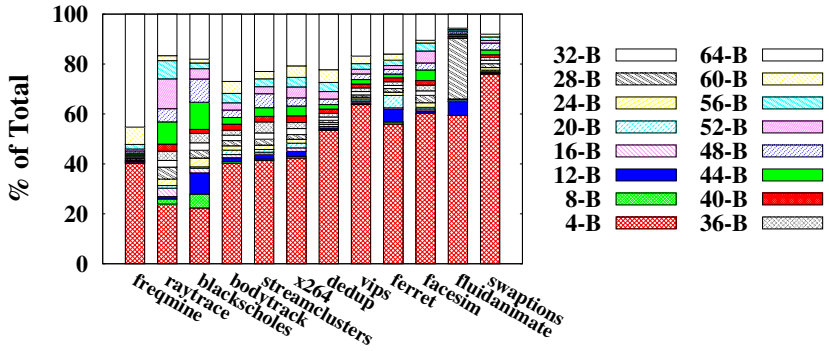
Figure 42(a) shows the average achievable compression ratios in each benchmark under the three different configurations. Even though the compression ratios vary with each benchmark (ranging from 0.63 to 0.17), the proposed DPC technique (SMA+FPA) always exhibits better compression ratios than each of the individual compression algorithms, as expected. Note that, as shown abstractly in Figure 32, the compressed DRAM cache benefits only from SMA compression (Phase 1), while the PCM device benefits from both SMA and FPA (after Phase 2 of the DPC).

Figures 42(b) and (c) show the breakdown of the sizes of the compressed cachelines observed in the DRAM cache and the PCM, respectively. As previously stated, the minimum unit of space for memory management is 4 B, for practical reasons. Thus, the size of a compressed cacheline can vary from 4 B (minimum possible size after compression) to 64 B (no compression). In fact, the information shown in Figures 42(b) and (c) is more insightful than the actual compression ratios shown in Figure 42(a). Over all benchmarks, the proportion of compressed cachelines compressed down to 4 B in the DRAM cache is significant, and this proportion does not change much after Phase 2. This means that even the simple SMA algorithm compresses most of the data in the DRAM cache very efficiently. However, we also observe the significant proportion of 64 B uncompressed cachelines in the DRAM cache (Figures 42(b)). This is where Phase 2 comes into play; these uncompressed cachelines are significantly reduced in the PCM after applying FPA compression in Phase 2 of the DPC mechanism (Figures 42(c)).

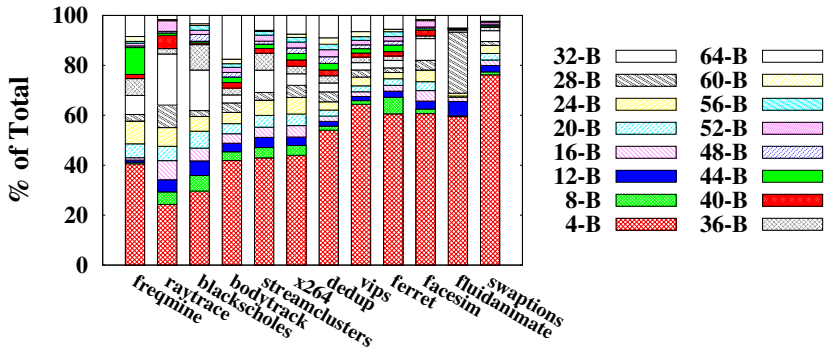
By distributing the compression process into two phases, the proposed DPC scheme can exploit the critical-word-first benefit in the DRAM cache. Figure 43 demonstrates this benefit in terms of average speedup. The speedup is normalized to a compression scheme combining the SMA and FPA algorithms in a *single* operation (as opposed to separating them in two phases, like in DPC) for fair comparison. The average speedup offered by DPC is 7%, while the speedups in Group 4 and Group 6 applications are 18% and 10%, respectively.



(a) Achievable compression ratios under SMA-only, FPA-only, and the proposed DPC, i.e., SMA+FPA (Lower is better).



(b) Breakdown of compressed cacheline sizes using SMA-only (i.e., after Phase 1 of the DPC).



(c) Breakdown of compressed cacheline sizes using SMA+FPA (i.e., after Phase 2 of the DPC).

Figure 42: Comparison of the achievable compression ratios over all benchmarks, and breakdown of the sizes of the compressed cachelines after Phase 1 and Phase 2 of the DPC scheme. Note that the closer the compression ratio is to zero, the better the compression efficiency.

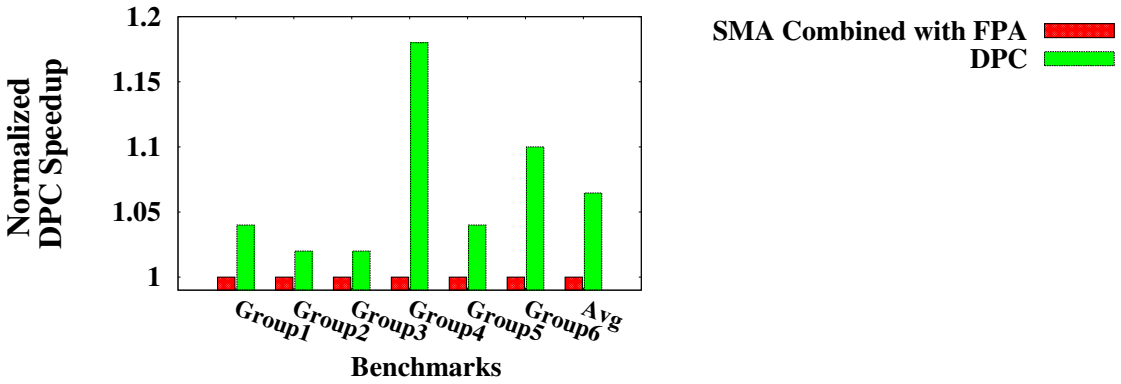
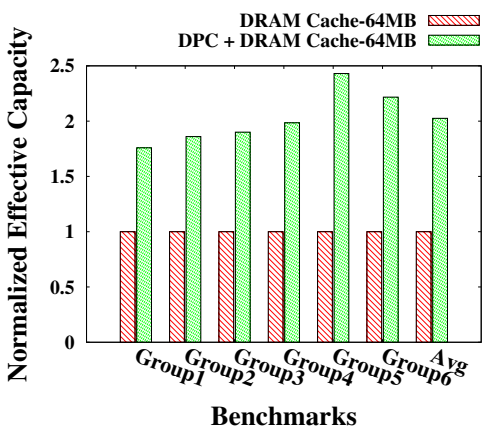


Figure 43: Normalized average speedup achieved by the proposed DPC scheme, normalized to the performance achieved by a compression scheme combining the SMA and FPA algorithms in a *single* operation.

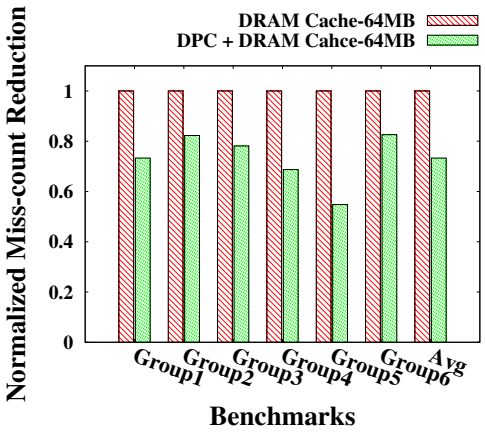
4.6.3 Overall Performance and Energy Improvements

We first measure the effective capacity and the reduction in the miss-count of the DRAM cache. These two attributes directly affect the performance and energy consumption of the entire main memory system. As shown in Figure 44, the proposed DPC mechanism significantly increases the effective capacity of the DRAM cache for all groups of benchmarks, which, in turn, reduces the miss-count of the DRAM cache. For example, in Group 5 benchmarks, the effective capacity is increased by 2.43 times, on average, during the application runs (Figure 44(a)). This reduces the number of misses by 42.2% (Figure 44(b)). To isolate the contribution of the DPC scheme from the effects of the cache management policy, we employ a conventional LRU replacement policy with a 64 MB DRAM cache for this evaluation.

Figure 45(a) shows performance comparisons between the four memory configurations under evaluation, as described in Section 4.6.1. We measured the time-to-completion of all benchmarks specified in each group, and normalized the results to the baseline configuration, i.e., the DRAM-only configuration, which is expected to yield the highest performance. Note that a higher bar indicates better performance. As expected, the proposed architecture – DRAM/PCM hybrid with DPC – shows the second-best performance for all groups of applications. This is because the DPC mechanism (1) significantly reduces



(a) Normalized effective capacity (higher is better).

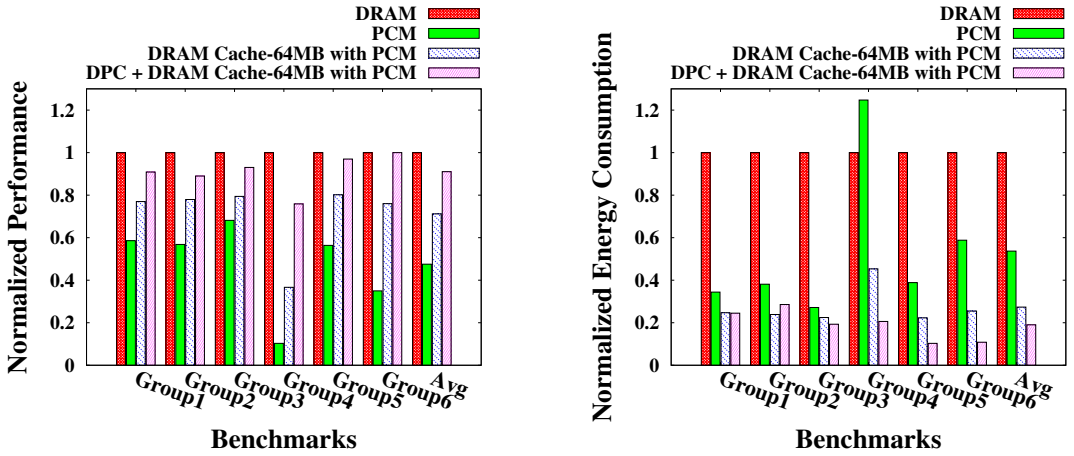


(b) Normalized cache miss-count reduction (lower is better).

Figure 44: Effective cache capacity and associated miss-count reduction in the DRAM cache. The results are normalized to a conventional 64 MB DRAM cache without DPC. The LRU replacement policy is used here.

the size of the evicted cacheline by efficiently reducing the actual size of data written to the PCM per every dirty miss, and (2) reduces the number of PCM access by increasing the effective DRAM cache size and reducing the miss count. On average, our proposed DRAM/PCM hybrid with DPC enhances the performance by 27.8%, as compared to the same hybrid configuration without the DPC mechanism.

We also compare the energy consumption of each configuration, as shown in Figure 45(b). Similar to Figure 45(a), the energy consumption for each configuration is normalized to that of the DRAM-only configuration. In this figure, a smaller bar indicates better energy efficiency. Since the DRAM-only configuration uses a large-size DRAM (8 GB), its energy consumption is substantially higher than the other configurations (more than 2 to 3 times), while the PCM-only configuration mostly consumes less energy than the DRAM-only configuration. The only notable exception is when executing Group 4 benchmarks, which are the most write-dominant applications, as indicated in Table 7. In this case, the PCM-only configuration actually consumes more energy than the DRAM-only configuration. If we focus on the two DRAM/PCM hybrid configurations, the DPC-augmented one



(a) Normalized performance (higher is better)

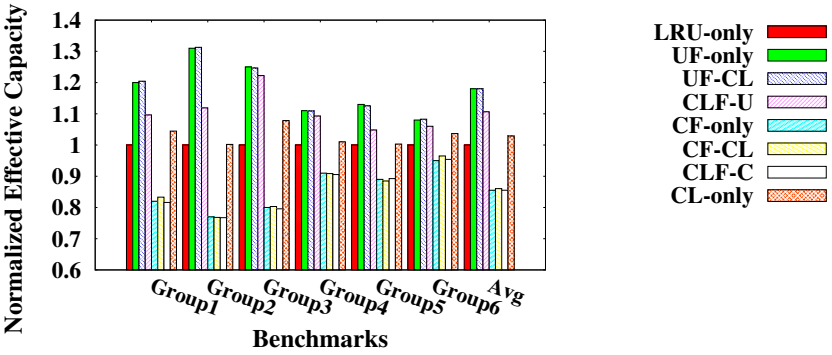
(b) Normalized energy (lower is better)

Figure 45: Performance and energy consumption results of the four memory configurations under evaluation. The results are normalized to the DRAM-only configuration, which is the reference point offering the best possible performance.

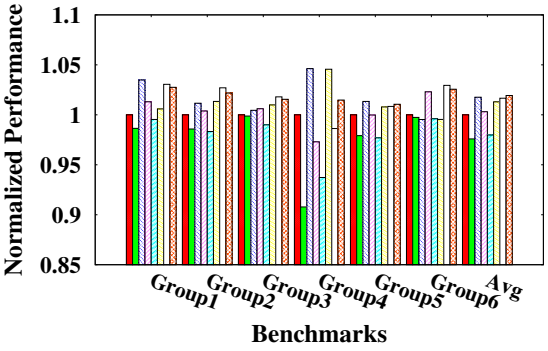
always consumes less energy than the other one. The reasons are similar to the ones described in our performance analysis: storing compressed data markedly reduces the number of energy-hungry PCM write operations. On average, the proposed DRAM/PCM hybrid with DPC enhances energy efficiency by 30.5%, as compared to the same hybrid configuration without the DPC technique. These results clearly demonstrate that the proposed DPC architecture can substantially outperform all other configurations, in terms of performance and energy consumption. This is a very attractive attribute, since future many-core systems will require both increased performance and superior energy efficiency, if they are to scale to tens, or hundreds, of cores.

4.6.4 DRAM Cache Design-space Exploration

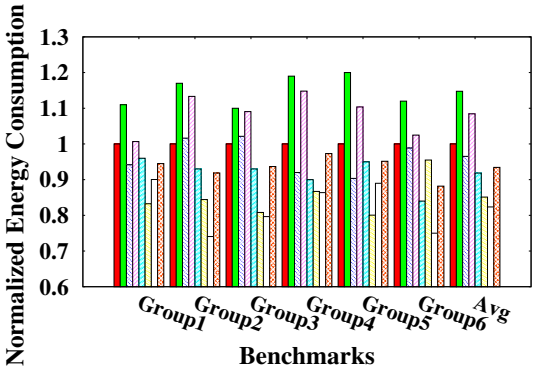
While the previous sub-section demonstrated the improvements emanating from the proposed DPC technique itself, this sub-section explorers the additional design-space of the DRAM cache by focusing on the compression effects within the DRAM cache, and the PCM device characteristics.



(a) Normalized effective DRAM cache capacity (higher is better).



(b) Normalized system performance (higher is better).



(c) Normalized energy consumption (lower is better).

Figure 46: Results under different cache replacement policies. Specifically, the impact on the effective capacity of the DRAM cache, the impact on overall system performance, and the impact on energy consumption are illustrated. All cases use the DPC scheme with a 64 MB DRAM cache, and the results are normalized to the LRU-only replacement policy. A total of 8 types of replacement policies are evaluated, based on the various combinations of priority levels employed by the *two-level priority mechanism* of Section 4.3.2.

4.6.4.1 DRAM Cache Replacement Policy

We first explore the design-space of the DRAM cache management policy, by concentrating on the cache replacement policy. A total of 8 types of replacement policies are evaluated, in terms of their impact on the DRAM cache's effective capacity, overall performance, and energy consumption. The 8 replacement policies are a result of the various combinations of priority levels employed by the *two-level priority mechanism* described in Section

4.3.2. Hence, the evaluated replacement policies are: (1) LRU-only, (2) Uncompressed-First-only (UF-only), (3) Uncompressed-First-CLean-second (UF-CL), (4) CLean-First-Uncompressed-second (CLF-U), (5) Compressed-First-only (CF-only), (6) Compressed-First-CLean-second (CF-CL), (7) CLean-First-Compressed-second (CLF-C), and (8) CLean-only (CL-only). Upon completion of the victim selection process by the priority-based policies described above, there may be multiple candidate cacheline victims. In such cases, the LRU status is used to select the least-recently used candidate for eviction. This LRU consideration aims to capture the locality behavior of the cachelines. These configurations relate directly to the cache replacement philosophies explained in Section 4.3.2.

The results are shown in Figure 46. All configurations use DPC with a 64 MB DRAM cache, and all values are normalized to the LRU-only configuration. Evicting the biggest cachelines (i.e., UF-only in the figure) always provides the biggest DRAM effective capacity (18% capacity enhancement, on average), as shown in Figure 46(a). However, by merely using the UF-only policy, the resulting DRAM capacity enhancements do not directly correlate to overall system performance, as illustrated in Figure 46(b). As previously explained in Section 4.3.2, this phenomenon is due to the fact that the eviction of the biggest cachelines significantly increases the PCM write delay and degrades the hit rate in the DRAM cache. Thus, the benefits of the increased DRAM effective capacity almost disappear, due to the additional PCM write delay. On the other hand, evicting the smallest cachelines (CF-only in the figure) degrades the DRAM effective capacity by 19.7%, on average, but it reduces the PCM write delay, because of the reduced data size sent to the PCM. In other words, the UF-only policy yields a positive effect on the DRAM cache (increased effective capacity), but it also yields a corresponding negative effect on the PCM. In the exact opposite manner, the CF-only policy yields a negative effect on the DRAM cache, but a positive effect on the PCM. As a result, both the UF-only and CF-only policies exhibit worse overall system performance than the conventional LRU-only policy.

As expected and described in Section 4.3.2, the configurations adopting multiple criteria – based on a two-level priority mechanism – show better performance than LRU-only for most applications, but the improvement magnitude varies with the applications. For example, the UF-CL policy (i.e., first-level priority is given to uncompressed lines, and the second-level priority is given to clean lines) effectively works for most benchmark groups (especially for Groups 1 and 4), except Group 6. Nevertheless, the performance when running Group 6 applications is still very close to the LRU-only policy. Note that Group 6 consists of the most compressible applications in the benchmark suite, which means that most cachelines in the DRAM cache are compressed. Thus, originally selecting a victim pool among uncompressed lines may not be so effective in this case. On the other hand, the policy works very well for Groups 1 and 4, which exhibit the highest and the second-highest memory access frequencies. These observations guide us to the conclusion that the UF-CL policy works very well for memory-intensive applications that are not so compressible.

The CLF-C policy (i.e., first-level priority is given to clean cachelines and the second-level priority is given to compressed lines), the CF-CL policy, and the CL-only policy also work relatively well with most groups of applications. Specifically, the CF-CL policy works particularly well for Group 4 applications, which are the most write-intensive benchmarks. In write-intensive applications, the probability of encountering dirty cachelines is relatively high, and this may result in unavoidable delays in the PCM (when dirty cachelines are evicted from the DRAM cache). Thus, for Group 4 applications, the filtering out of the dirty cachelines by the CLF-C policy is not very effective and does not help the PCM. On the contrary, the CF-CL policy – which evicts the smallest-sized (compressed) data first – is more effective than the CLF-C policy in minimizing the heavy PCM write delay.

Based on these observations, it becomes very clear that considering only one factor in the DRAM cache replacement policy does not help in enhancing the overall system performance. However, if one considers two factors – based on the proposed dual-level priority

mechanism – performance can be markedly improved. We also found that minimizing the evicted data size to the PCM affects the overall performance of the memory system more pronouncedly than enhancing the effective capacity of the DRAM cache. This is a direct consequence of the heavy write delays currently exhibited by PCM technology. However, we expect that the impact of enhancing the effective capacity of the DRAM cache will become increasingly more important in the future, if technology improvements lead to a decrease in the PCM's write latency.

From an energy perspective, the trends are more straightforward and easy to decipher, as illustrated in Figure 46(c). Since most energy is consumed during the PCM write operations, reducing the absolute amount of evicted data to the PCM is the most important factor pertaining to energy consumption. As a result, the UF-only policy results in the highest energy consumption (almost 14.8% higher energy consumption than that of the LRU-only policy, on average), while the CF-CL and the CLF-C policies show the lowest energy consumption (almost 14.9% to 17.7% less energy consumption than LRU-only, on average). Although the UF-CL policy does not show better energy efficiency than the CF-CL or the CLF-C policies, its energy consumption is still lower than LRU-only (by approximately 3.5%).

The final conclusion of this sub-section is that, unfortunately, there is no clear-cut winner in terms of system performance. The best-performing policy varies with the application type, and it does not always yield the lowest energy consumption. Hence, the designer must choose the DRAM cache replacement policy that best satisfies the required system design goals, based on the application characteristics. The decision must strike the best performance/energy tradeoff, as dictated by the system requirements.

4.6.4.2 DRAM Cache Size

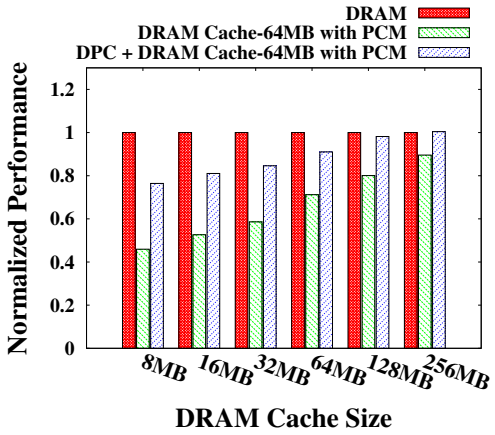
Another important factor affecting the performance and energy consumption of the memory system of DRAM/PCM hybrids is the DRAM cache size. Therefore, we explore the impact of the DRAM cache size on two configurations: a DRAM/PCM hybrid without DPC,

and an equivalent hybrid setup employing the proposed DPC mechanism. The results are shown in Figure 47. All parameters, except the DRAM cache size, are fixed. The size of the DRAM cache is varied from 8 MB to 256 MB, and the obtained performance and energy consumption results are normalized to a DRAM-only configuration. As shown in Figure 47(a), the performance degradation of an 8MB DRAM cache without DPC is almost 54.1%, as compared to the DRAM-only configuration, while that of an 8MB DRAM cache with the proposed DPC mechanism is only 23.6%. The rate of the performance degradation is reduced by increasing the DRAM cache size. When the DRAM cache size exceeds 128 MB, the DPC-augmented configuration shows almost identical performance with the DRAM-only configuration. In terms of energy consumption (Figure 47(b)), both DRAM cache configurations exhibit significantly lower consumption than the DRAM-only configuration. These two observations corroborate our claim that the DPC mechanism is ideal in maximizing the performance of DRAM/PCM hybrids (potentially equalling the performance of DRAM-only systems), while also minimizing the total energy consumption.

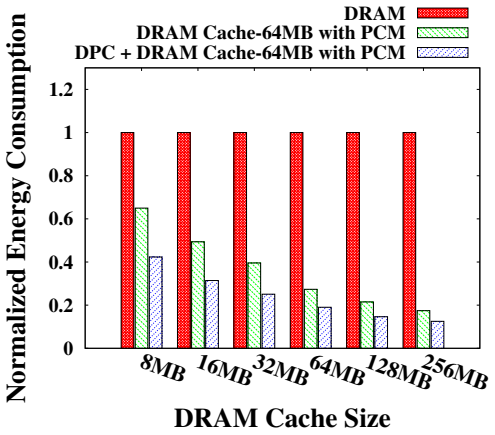
4.6.5 Durability and Lifetime of the PCM Device

Long-term endurance is regarded as the most critical problem in PCM-based memory architectures, because it is directly related to system reliability. In order to correctly estimate the endurance of the memory configurations under evaluation, we first run each benchmark and track the number of writes during execution. Only one counter per each 64 B unit of the PCM is used (i.e., the equivalent of a 64 B cacheline in the DRAM cache). Note that this cache-line-based counting method does not diminish the accuracy of our endurance estimation, because the minimum unit of write operation is exactly the same as the counting granularity. This means that all PCM cells in one 64 B unit will experience the exact same number of writes.

After completion of execution on the simulated machine, we compare all the write counters and find out the 64 B unit with the maximum write count. Based on the total execution time of the benchmark and the maximum write-count number during that



(a) Normalized system performance (higher is better).



(b) Normalized energy consumption (lower is better).

Figure 47: Performance and energy consumption results as the DRAM cache size of DRAM/PCM hybrid systems is varied. The results are normalized to the DRAM-only configuration.

time, we can estimate the lifetime of the PCM device. We use 10^8 as the maximum number of writes that can be performed to any cell before the cell fails [83]. In total, five different configurations are used to break down the contribution of each technique in detail: Hybrid DRAM/PCM without compression and without any wear-leveling technique, Hybrid DRAM/PCM without compression and normal Page-level Swapping (PS), Hybrid DRAM/PCM with DPC + PS, Hybrid DRAM/PCM with the Compression-based Segment Rotation (CSR) technique and DPC + PS, and Hybrid DRAM/PCM with a small Local Counter (LC) with CSR + DPC + PS.

Figure 48 compares the PCM lifetimes of the five configurations. The PCM lifetime for each configuration is normalized to that of Hybrid DRAM/PCM without compression and without any wear-leveling technique. The DPC + PS with Hybrid DRAM/PCM configuration enhances the lifetime of the PCM by up to 6.6 times. If the CSR technique (Section 4.4.1) is applied to the DPC + PS with Hybrid DRAM/PCM, the lifetime is enhanced by up to 16.8 times, as compared to the baseline. After adding the LC-based mechanism (Section 4.4.2), the lifetime of the PCM is enhanced by up to 43.1 times, as compared to the baseline configuration.

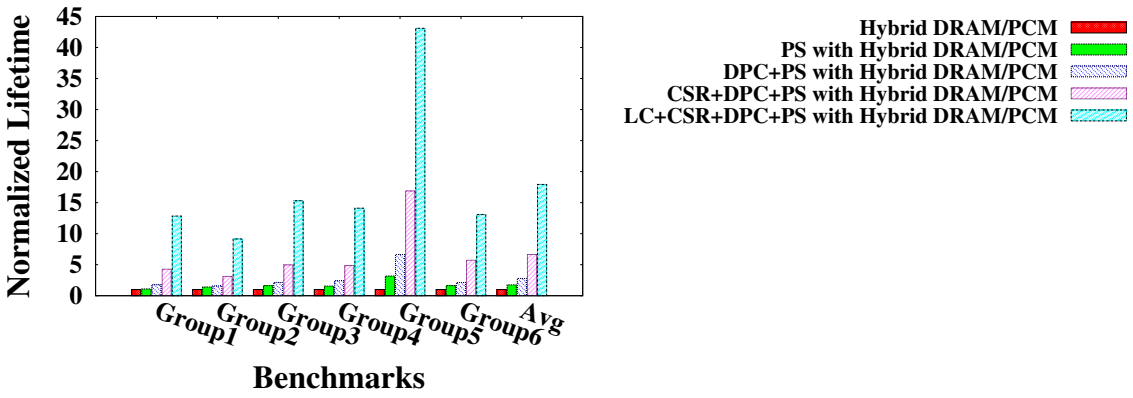


Figure 48: Comparison of PCM lifetime when using different wear-leveling techniques. The PCM lifetime for each configuration is normalized to that of a hybrid DRAM/PCM without compression and without any wear-leveling technique.

In this evaluation, we do not directly compare the lifetime enhancement of our technique with the various existing wear-leveling techniques, because the proposed multi-faceted wear-leveling technique can still coexist with other wear-leveling mechanisms to further prolong the PCM lifetime. As previously mentioned, the key benefit is that – unlike other similar techniques – the multi-faceted wear-leveling mechanism presented in this work requires minor modifications only at the memory controller level (not at the device/architecture levels).

4.7 Conclusion

The abundance of on-chip processing capability necessitates an equally capable memory sub-system, which can keep up with the microprocessor. Phase-Change Memory is emerging as a viable alternative to DRAM for the off-chip main memory of future CMPs, because of attractive qualities, such as lower power consumption and inherent ability to scale down to minuscule sizes. However, PCM's slow write performance and limited durability necessitate the employment of additional supporting solutions to assist with the memory operations. One popular technique is the hybridization of PCM and DRAM technology in a memory architecture that uses the DRAM as an off-chip cache to the much higher-capacity PCM.

In this chapter, we proposed a novel enhancement mechanism for such DRAM/PCM hybrids, and explore its design-space to maximize the reaped benefits. The Dual-Phase Compression (DPC) scheme uses two complementary compression algorithms to achieve very efficient compression ratios. By increasing the effective capacity of the DRAM cache, the DPC-enhanced architecture significantly reduces the number of PCM accesses, as well as the average size of access, by storing compressed data. This, in turn, yields great performance and energy consumption improvements. The proposed technique is specifically tailored for PCM-based systems and it is architected in such a way as to be a self-contained and self-supported solution, which does not require any support from – or any modification to – the various layers of the cache/memory hierarchy.

Additionally, the DPC mechanism incorporates a multi-faceted wear-leveling process that intelligently balances wear-out within the PCM, in order to prolong the lifetime of the device. This wear-leveling mechanism requires only minimal modifications at the memory-controller level. Hence, the design presented in this work addresses a triptych of valuable metrics simultaneously: performance, energy, and long-term durability.

In order to evaluate the proposed DPC architecture, we employ a detailed simulation framework driven by traces extracted from a full-system simulator running real applications. The dual-phase mechanism is demonstrated to achieve 27.8% performance improvement and 30.5% energy reduction, on average, as compared to a baseline DRAM/PCM hybrid implementation. The benefits of the DPC mechanism are maximized by further exploring the design-space of the DRAM cache. By focusing on the cache management policy, we demonstrate additional performance improvements of up to 4.7%, and additional reduction in energy consumption of up to 19.9%. Finally, the multi-faceted wear-leveling technique is shown to significantly prolong the lifetime of the PCM to well beyond the typical useful lifetime of a computer system.

CHAPTER 5

COMPRESSION-BASED HYBRID MLC/SLC MANAGEMENT TECHNIQUE FOR PHASE CHANGE MEMORY SYSTEMS

Over the last few years, several new memory technologies have been announced, in an effort to address some of the shortcomings of DRAM technology. One of the most promising new actors is Phase-Change Memory (PCM), which is gaining a foothold in the research community, predominantly due to its ability to scale very deeply into the low nanometer regime [19]. PCM exploits the unique trait of chalcogenide glass to switch between two states – crystalline and amorphous – with the passage of an electric current. However, PCM is marred with some problematic inherent characteristics, such as extremely poor write performance and fairly limited long-term endurance, as compared to the venerable DRAM technology. Researchers have attempted to tackle these show-stopping artifacts by proposing a wide spectrum of solutions, ranging from the device level all the way to the memory system architecture level.

In a most recent development, researchers have demonstrated the feasibility of Multi-Level Cell (MLC) arrays in PCM [26, 27, 84, 85, 86, 87, 88, 89, 90]. Although this MLC technology doubles, or even quadruples, the capacity benefits of PCM, it does so at the expense of much longer latencies and decreased cell endurance, when compared to Single-Level Cell (SLC) PCM devices. Hence, in order to enjoy the vast capacity potential of MLC PCM, more concrete and sophisticated ways to compensate for the deteriorated performance and durability must be devised.

Several system-level architectural solutions have been proposed to enhance MLC-based PCM memory systems by adaptively changing their storage mode between SLC and MLC. However, most of these techniques require either complicated a priori workload profiling [91], or infeasible real-time tracking of memory utilization to determine the configuration

mode [3]. In addition, address remapping due to the dynamically changing memory capacity is required, which complicates the solution even more. Recently, a dynamic MLC management scheme considering frequent zero-values has been proposed [78]. The scheme exploits the reduced data size after compression. Said work embeds all required logic inside the PCM device itself, so as to cooperate directly with the MLC PCM cells. This design decision results in some practical drawbacks, as will be explained later on. Moreover, the aforementioned technique requires non-negligible metadata space (almost 9% area overhead) for storing the coding and merging bits [78].

In this chapter, we introduce a lightweight, yet powerful, mechanism to **dynamically re-configure the memory space between SLC and MLC arrays, *without requiring any profiling, nor address re-mapping***, and with modifications required only within the PCM memory controller. While the basic philosophy of existing proposals pertaining to MLC PCM are relying on *memory utilization or required capacity*, our scheme exploits a simple *compression* technique in order to reduce the size of the data itself. The proposed concept derives from the fact that 2-bit MLC PCM can store two bits in a cell. Hence, the same amount of data can be stored in *half* as many PCM cells, when MLC PCM mode is used. Note that MLC-capable cells can also be used in SLC mode (i.e., to store only a single bit per cell). When used in SLC mode, these cells are accessed (read and written) much faster than when in MLC mode. In fact, they can be accessed as fast as normal, conventional PCM cells that operate only in SLC mode [3]. This realization is the fundamental driver in our work: compress the data size appropriately, so that some information can be stored in SLC mode (i.e., lower storage density, but much faster performance). Since the storage density of MLC mode is *double* that of SLC mode, if the compressed data size is less than 50% of its original size, the *compressed* data can fit into the same number of memory cells, but in SLC mode (i.e., at *half* the storage density). If the 50% compression threshold is not satisfied, the data is stored in the high-density (and slower) MLC mode.

Compression techniques have generally been used to increase the effective capacity of

the memory/storage space. However, they require the use of an address remapping scheme, as well as somewhat complex compaction and/or re-allocation processes, because of the variable size of the compressed data. In this work, we specifically aim to minimize the overhead associated with traditional compression techniques. Toward this end, we design a memory system that can adaptively and dynamically re-configure the PCM space into SLC or MLC modes efficiently.

In summary, the main contributions of this work are:

- The design of a PCM memory system architecture that dynamically configures the memory space into SLC or MLC modes *without requiring any complex application profiling, or any address remapping*.
- The proposed compression technique's purpose is two-fold: to enable effective and efficient MLC/SLC dynamic re-configuration, but also to *reduce memory traffic and latency in general*, regardless of whether the data space is configured as SLC or MLC.

Our assertions are validated through trace-driven simulations of real application workloads. The results demonstrate (a) the efficacy of the proposed compression-based MLC/SLC management technique, and (b) the viability of MLC PCM in future computer systems.

The rest of the chapter is organized as follows: Section 5.1 serves as a preamble by introducing the basic principles of MLC PCM technology. Section 5.2 then proceeds with the conceptual and architectural design of the proposed compression-based MLC/SLC PCM memory management technique, while Section 5.3 evaluates its effectiveness. Finally, Section 5.4 concludes the chapter.

5.1 Background on Multi-Level PCM technology

PCM technology exploits the resistance differences between the amorphous state (RESET value, high resistivity) and crystalline state (SET value, low resistivity) of chalcogenide

glass. The resistance in the amorphous state is generally 3 or 4 orders of magnitude larger than the resistance in the crystalline state [74]. This wide range of resistance *differences* between the RESET and SET states has given rise to MLC PCM technology. The ITRS roadmap [6] projects that 4 bits per PCM cell will be possible by 2012. However, MLC operations need dedicated control steps for the resistance distribution during write operations, and then it must precisely distinguish between these different resistance levels during read operations.

For write operations, Write-and-Verify (WAV) is a well-proven technique for distributing the resistance in MLC PCM devices [27]. It repeatedly writes and verifies cells until it achieves the required target resistance. The required number of WAV iterations is generally proportional to the number of bits per cell. Hence, writing to MLC cells consumes more energy and incurs a higher latency. This, in turn, degrades the already limited lifetime of PCM cells. Reading data from MLC cells also incurs additional latency – as compared to reading data from SLC PCM – because the process involves more comparison steps.

Table 8 compares the latency and endurance attributes of SLC and MLC PCM devices [3]. Since the number of iterations in the WAV process is assumed to be four when storing two bits per MLC cell, the read and write latencies of MLC cells increases by almost 4 times, as compared to SLC cells. Although the endurance of MLC PCM cell is degraded as well, this degradation is overshadowed by the degradation in latency, which has a more profound impact on the use of MLC PCM in modern computer systems. This degradation in access latency will be used when comparing the memory system performance in Section 5.3.

Table 8: Comparison of the main SLC and MLC PCM attributes [3]

	SLC	MLC
Read latency	10 ns	44 ns
Write latency	100 ns	395 ns
Endurance (Reset cycles)	10^9	10^7

5.2 Compression-based Adaptive MLC/SLC PCM Management

Compression techniques are attractive within the context of PCM, because they reduce the number of PCM accesses after compression. This is very useful in enhancing both the performance and endurance of the PCM. Furthermore, the remaining space left after the compression can be used to further enhance performance and endurance, without incurring the overhead of address remapping or re-allocation. In this work, we exploit the reduced (compressed) data size to form an adaptive hybrid MLC/SLC PCM, which provides SLC-like performance at MLC storage capacity. Without loss of generality, this work assumes 2-bit MLC (i.e., 2 bits are stored in one MLC cell). However, a similar methodology can be applied to higher-density MLC cells with minimal parameter modifications.

5.2.1 Adaptive MLC/SLC PCM Re-configuration

The main idea of this work is to exploit compression techniques to adaptively change the storage mode between SLC and MLC, without incurring severe overhead. Figure 49 depicts a high-level, conceptual view of the idea we propose. Basically, we assume that the main memory space consists of n-byte unit blocks, and all memory blocks are statically addressed based on MLC capacity. This means that one memory block physically comprises $\frac{n}{2} \times 8$ cells. If this block is set in SLC mode, the same block can only store $\frac{n}{2}$ bytes. The parameter n refers to the cacheline size, in bytes, of the on-chip Last-Level Cache (LLC), and this is conventionally the minimum access unit of most main memory systems. As such, all compressions and MLC/SLC mode changes are performed at the granularity of this particular unit size. Note that, in order to avoid the complex address remapping problem, we do not use the remaining space – marked as "Unused cells" in the figure – for storing extra data blocks. Instead, we exploit the remaining space to adaptively change the storage mode to SLC for further enhancement of the performance and long-term endurance.

If an n-byte block can be compressed to a size of $n/2$ (or less), then it can fit in the same physical space in the PCM, but in SLC mode (i.e., at *half* the storage density of MLC). If stored in SLC mode, the block can be accessed much faster (four times as fast, as

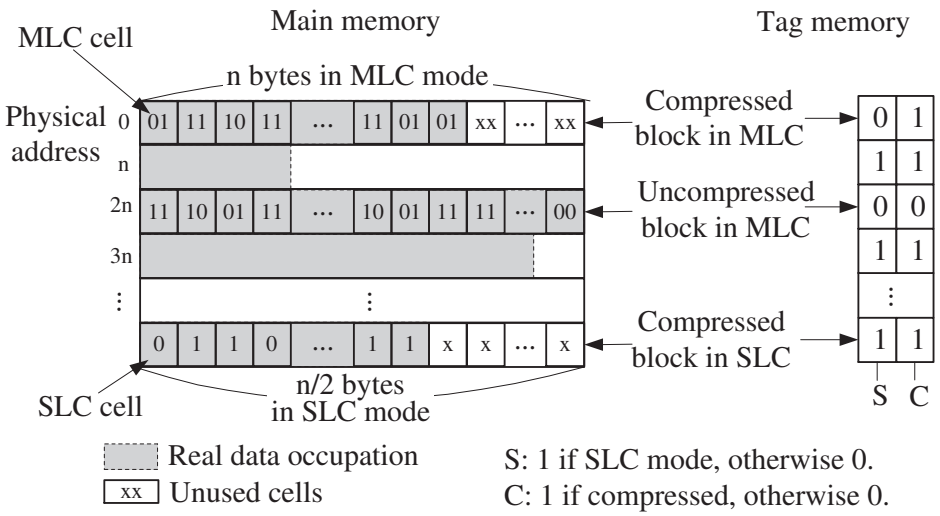


Figure 49: A high-level overview of the proposed adaptive MLC/SLC mode reconfiguration. Since the storage density of MLC mode is *double* that of SLC mode, if the compressed data size is less than 50% of its original size, the *compressed* data can fit into the same number of memory cells, but in SLC mode (i.e., at *half* the storage density), as shown in the last row of the diagram. In SLC mode, $n/2$ bytes can fit in the same space that could potentially hold n bytes in MLC mode.

described in Section 5.1), while still occupying the same physical space as an n-byte block stored in MLC mode. For efficient use of storage space without any loss of information, we define three types of block configurations, depending on the *compression ratio*[1] achieved: (1) Compressed block in SLC mode, when the compression ratio is lower than 0.5[2], (2) Compressed block in MLC mode, when the ratio is higher than 0.5, but lower than one, and (3) Uncompressed block in MLC mode for all other cases. As shown in Figure 49, this classification only requires 2 extra bits per data block, which corresponds to 0.2% of the total storage space (assuming n is 128 bytes).

If the compression ratio of the block is lower than 0.5 (i.e., less than $n/2$ bytes of space are required after compression), this block can be stored in SLC mode without any loss of information. In this case, performance can be dramatically increased, because read/write operations can be done in the fast SLC mode. If the compression ratio is higher than 0.5,

[1]The *compression ratio* is defined as $\frac{compressed\ size}{original\ data\ size}$; thus, a *lower ratio is better.*

[2]We set this threshold value to 0.5, because we assume 2-bit MLC technology. In the case of 4-bit MLC technology, this threshold would be set to 0.25.

the block should be stored in MLC mode, since it cannot fit in an SLC-configured block. Even in this case, however, we can still reduce the number of memory access operations if the compression ratio is lower than one, since the size of the compressed data is still smaller than the original size.

Overall, by adaptively configuring each block as SLC or MLC based on the compression ratio, our technique can efficiently enhance the performance of the memory system while still utilizing the same capacity of MLC-only mode configurations.

The main benefit of the proposed compression-based MLC/SLC re-configuration technique is two-fold:

- Unlike previous approaches [91, 3] that require address remapping and/or application profiling for adaptive MLC/SLC switching, our approach simply decides the operating mode based solely on compression ratio. Furthermore, this scheme does not require any address space remapping, despite the fact that a conventional compression technique is used. Even though the memory space configured in SLC mode loses half of its effective capacity (as compared to MLC mode), we do not need to remap the memory address, because the required memory capacity is also reduced to less than half of its original capacity due to the compression.

- The compression effect also reduces the actual amount of data to be read or written. This may not be so effective in increasing *read* performance, but it is quite helpful in enhancing *write* performance. This is because the read operations in PCM cells can be performed in parallel (normally several thousands of PCM cells), while the write operations must be performed almost serially, because the number of PCM cells that can be written in parallel are limited to just a few cells at a time, due to heavy power consumption [92].

Of course, **the effectiveness of this whole concept depends on the overhead incurred by the compression scheme**. Hence, we proceed by first analyzing how the compression

algorithm affects the performance enhancement of our proposed adaptive MLC/SLC management technique in the following sub-section. We then present the architectural support required to realize the proposed idea in feasible implementation.

5.2.2 Compression Algorithms

The compression ratio achieved is one of the most important factors in determining the overall memory system performance, because a better compression ratio translates to more blocks that can be configured in SLC mode. In general, there is a trade-off between the compression overhead and the achievable compression ratio. We evaluate two conventional compression algorithms: Successive Matching Compression (SMC) – as an example of a simple word-level compression algorithm with very lightweight implementation cost – and Frequent Pattern Compression (FPC) [79], as an example of a more complex bit-level data compression that can achieve better compression. The SMC algorithm compares the current word with its successive word, and if the two words are exactly the same, a corresponding compression tag bit (just one bit) is set and the second word is eliminated. On the other hand, FPC uses a pre-defined Frequent Pattern Encoding Table in order to achieve better compression ratios, but it requires substantially more hardware resources than the simple SMC algorithm. Note that the compression algorithm itself is not claimed as a contribution of this work. We simply analyze how existing compression algorithms affect our adaptive MLC/SLC re-configuration technique.

Figure 50 presents the compression ratios and the proportion of memory blocks configured in MLC mode when using the SMC and FPC algorithms. For this experiment, various applications of the PARSEC benchmark suite [2] are executed in a trace-driven simulation framework involving an 8-core Chip Multi-Processor (CMP). The details of the evaluation platform are presented in Section 5.3. With the exception of "canneal," all other applications exhibit compression ratios below 0.4, even under the simple SMC algorithm. The MLC-configured block ratio is even lower than the corresponding compression ratio. This means that more than 60% of the PCM memory space can be configured in fast SLC mode,

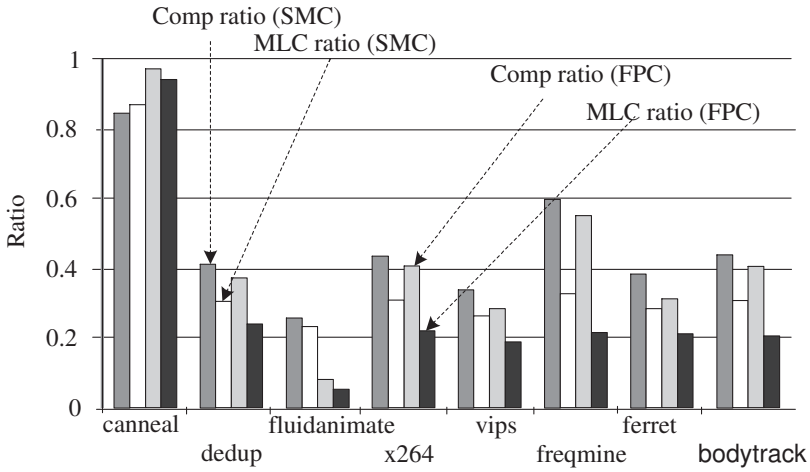


Figure 50: Compression ratio (lower is better) and MLC-configured block ratio (lower is better), when using various applications from the PARSEC benchmark suite [2] in a trace-driven environment simulating a 16-core Chip Multi-Processor (CMP).

rather than the slow MLC mode.

The timing and implementation overhead of the two algorithms is presented in Table 9. As expected, since FPC is a more complicated compression algorithm, its overhead is higher than the overhead of SMC. However, both algorithms incur reasonable overhead that can potentially be tolerated by the system designer. Note that for the fair comparison of the two compression algorithms, the timing overhead is included in all our evaluations.

5.2.3 Architectural Support and Implementation

In order to efficiently support our proposed compression-based MLC/SLC re-configuration technique, we assume the memory system architecture depicted in Figure 51. It consists of two main parts: a PCM memory controller and the PCM device itself. The compression and

Table 9: Timing and implementation overheads of SMC and FPC.

Overhead		SMC	FPC
Timing (cycles)	Compression	1	3
	Decompression	1	5
Area	Gate count	394	8,758
	%[1)]	0.5	10.4
Power (mW)		1.02	4.07

[1)] % of a commercial memory controller

de-compression engines, as well as the additional tag memory, are embedded into the PCM memory controller. This is one major difference with the technique of [78], which embeds all compression logic inside the memory device itself. Incorporating the compression logic within the memory device has the following possible drawbacks: (1) resource duplication (all memory devices must have the same compression tables/logic), (2) management of the compression parameters, e.g., the Frequent Value (FV) table, is non-trivial, and (3) all tag bits must be stored in the PCM device at a fixed position, which affects the wear-leveling mechanism. To avoid these limitations, we embed the compression engines into the memory *controller*.

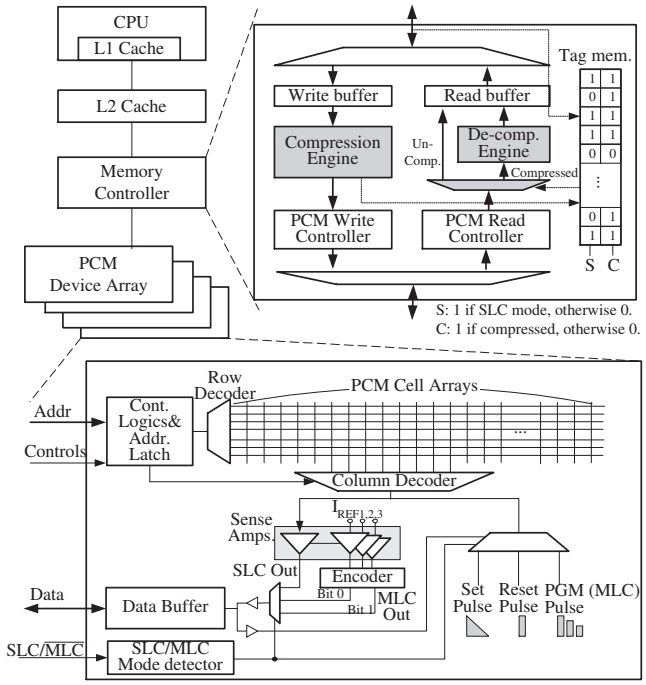


Figure 51: The PCM memory controller and device architecture assumed in this work.

The additional tag memory required (2 bits per cache-line-sized memory block) can be calculated as $2\ bits \times \frac{Entire\ Memory\ Capacity}{Cache\ Line\ Size}$. For example, a 1 GB memory space configured to use 128 B cachelines in the LLC requires merely 2 MB of additional tag memory space.

The algorithms employed during the *read* and *write* operations are presented in detail in Figure 52 and Figure 53, respectively.

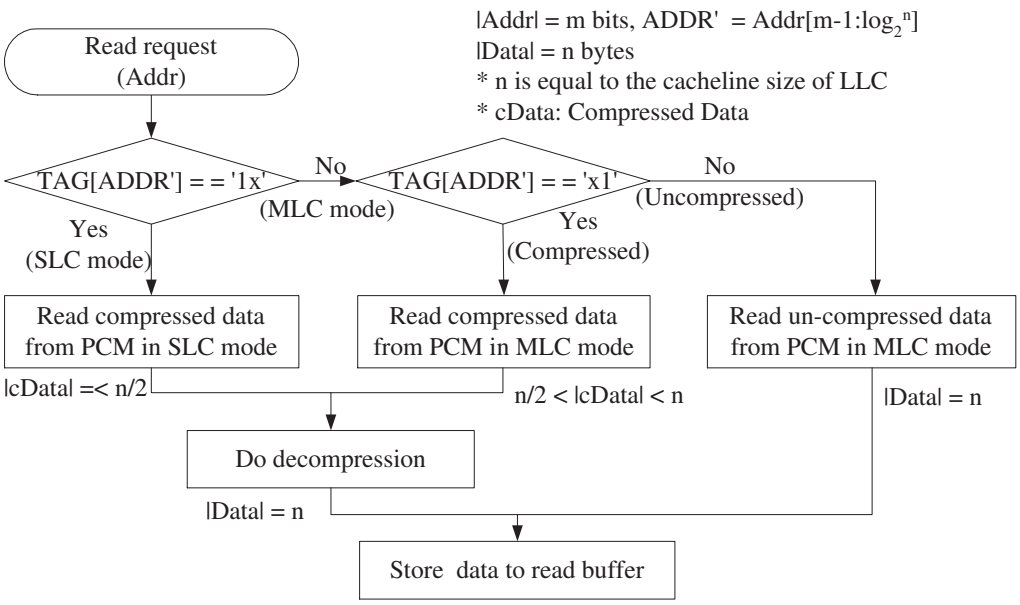


Figure 52: The algorithm employed during *read* operations.

When a read request arrives with its corresponding physical address from the LLC, the PCM memory controller first checks the block's corresponding S bit (status bit) stored in the tag area. If this bit indicates that the data is stored in SLC mode, then the memory controller sends appropriate SLC read signals to the PCM, and then read the compressed data from the PCM. Finally, the decompression engine decompresses the data to its original size and stores it to the read buffer. On the other hand, if the corresponding S bit is set to '0' (i.e., MLC mode), the memory controller reads the data from the PCM in MLC mode. Depending on the state of the C bit (compression bit), the controller decompresses the compressed data (*cData*) and stores it to the read buffer, or directly stores the uncompressed data to the read buffer.

Write operations happen in a reverse manner. When a write request arrives with its address and data from the LLC, the controller first compresses the data, and then checks the compression ratio. If the compression ratio is lower than 0.5, the controller sets the block's corresponding tag bits (S and C) to '11', and writes the compressed data to the PCM in SLC mode. If the compression ratio is more than 0.5, but still smaller than one, the controller writes the compressed data to the PCM in MLC mode while setting the corresponding tag

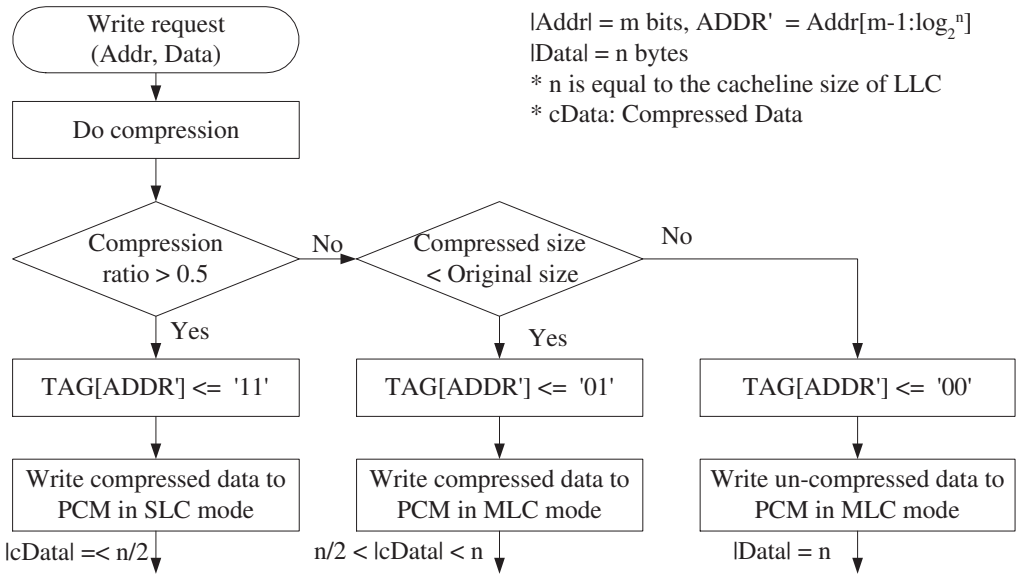


Figure 53: The algorithm employed during *write* operations.

bits to '01'. If the compression ratio is larger than one (i.e., the compressed data size is bigger than the original data size), the uncompressed data is written to the PCM in MLC mode, with the corresponding tag bits set to '00'.

The PCM device architecture required to support the proposed MLC/SLC adaptive reconfiguration technique is shown at the bottom of Figure 51. Our device architecture is based on a modified version of the adaptive MLC/SLC PCM array structure of [3], which supports SET, RESET, and Partial Set Pulses for MLC/SLC PCM cell writing, and dual mode sense amplifiers for MLC/SLC reading.

5.3 Experimental Evaluation

5.3.1 Simulation Framework

To evaluate the proposed technique, we employ an in-house developed trace-driven simulator to conduct various experiments. The traces are extracted from real applications running on the Simics full system simulator [40]. The details of the simulation framework are described in Table 10. Several representative applications are selected from the PARSEC benchmark suite [2].

Table 10: Simulated system parameters.

Number of CPU Cores	8
CPU Core Type	UltraSPARC-III+, 2 GHz
L1 caches (Private)	I- and D-cache: 32 KB, 4-way 128 B block
L2 caches (Private)	Unified 512 KB, 4-way 128 B block
Main memory	8 GB PCM (SLC and MLC)

The performance and energy consumption of three types of main memory configurations are evaluated: (1) a baseline configuration with an 8 GB PCM module (in MLC-only mode, designated as *MLC_Only*); (2) a system with MLC-only PCM that uses memory compression (*MLC_Comp*); and (3) our proposed MLC/SLC adaptive re-configuration setup with memory compression (*MLC/SLC*). The two previously explained compression algorithms (SMC and FPC) are used for comparison purposes. The latency and energy values for the SLC and MLC PCM devices are derived from existing literature [3, 93]. Note that all the simulations include the overhead incurred by the compression/decompression techniques, as depicted in Table 9.

5.3.2 Evaluation Results

We first evaluate the performance of each configuration, as shown in Fig. 54(a). The performance is normalized to the baseline configuration (*MLC_Only*). As expected, the performance enhancement is mainly proportional to the compression ratio. Compression with the FPC algorithm shows better performance than with SMC. Significant improvements are observed in most applications, except "canneal." Said application is not easily compressible – even with FPC – with only 15% of data being stored in SLC mode. If only compression techniques are applied (*MLC_Comp*), without the proposed MLC/SLC management scheme, performance with SMC and FPC still improves by 2.0 times and 2.7 times, respectively. The addition of the proposed adaptive MLC/SLC management technique achieves, on average, 2.7 times (with SMC) and 3.6 times (with FPC) performance enhancement.

Similar observations can be made when evaluating the energy consumption, as shown in Fig. 54(b). The adaptive MLC/SLC management technique lowers energy consumption

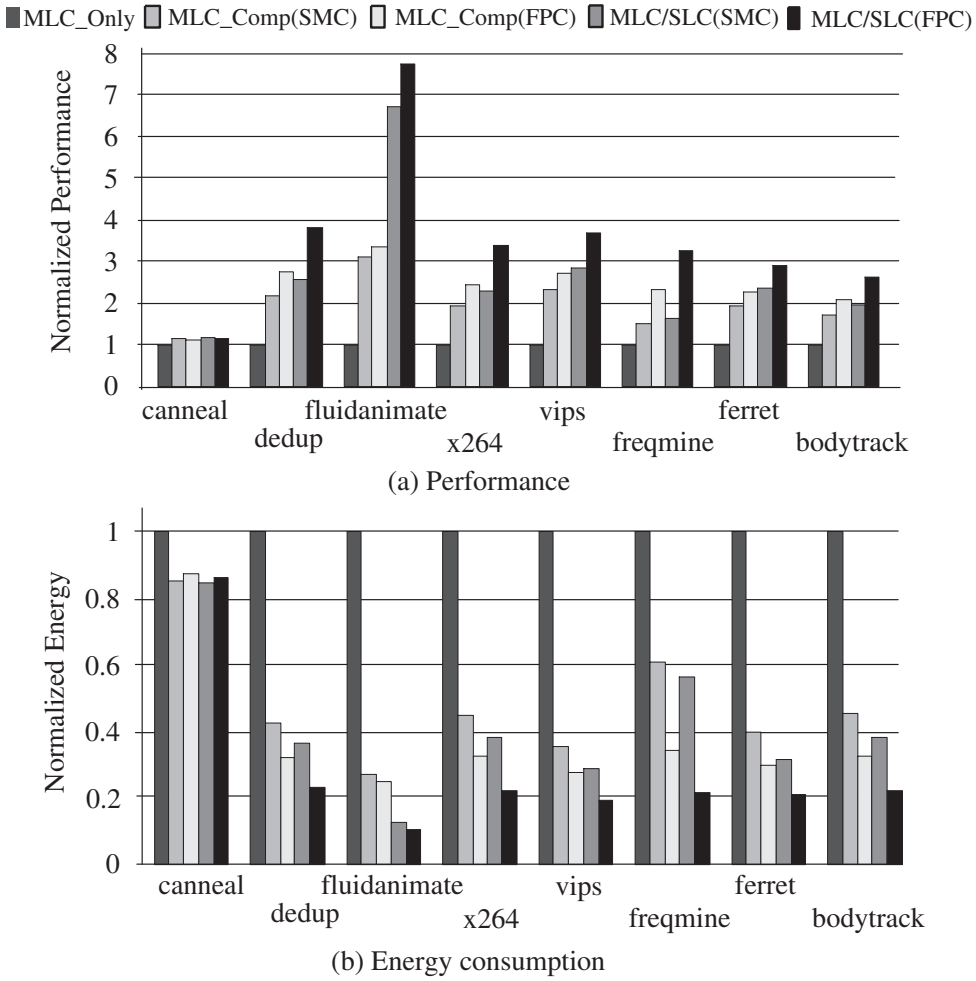


Figure 54: Comparison of (a) Performance and (b) Energy consumption for the various configurations. The results are normalized to the MLC_Only configuration.

by an average of 59% with SMC, and 72% with FPC, as compared to the *MLC_Only* mode. Note that *these improvements do not rely on any dynamic application profiling of memory behavior and/or utilization of memory space*.

5.3.3 Implications on Device Lifetime

Although we do not provide any sophisticated wear-leveling techniques and analysis (which is outside the scope of this work), we can still *estimate* the relative long-term endurance of our proposed scheme, as compared to an MLC-only PCM mode. The estimate is extracted from the percentage of memory blocks stored in SLC mode (see Fig. 50) and the parameters in Table 8, while assuming that the same wear-leveling algorithms are used as

in the MLC-only case. The effective endurance of the proposed MLC/SLC adaptive re-configuration architecture would enhance the long-term endurance by approximately 5.8 times (with SMC) and 7.1 times (with FPC), as compared to an MLC-only configuration.

5.4 Conclusion

Multi-Level Cell (MLC) technology *doubles* the storage capacity of PCM-based main memory systems at the expense of much longer latencies and decreased cell endurance, as compared to Single-Level Cell (SLC) PCM. In this work, we strive to mitigate these inherent MLC deficiencies, in order to effectively tap the vast potentials of MLC PCM technology. We propose a compression-based adaptive MLC/SLC re-configuration technique, which combines the performance benefit of SLCs with the higher capacity of MLCs. The key advantage of the proposed mechanism – as opposed to prior adaptive MLC/SLC techniques – is that it does not require any complex dynamic application profiling, or address remapping. Trace-driven simulations with real applications demonstrate that the proposed architecture yields 3.6 times system performance improvement, while lowering energy consumption by 72%, on average, as compared to MLC-only PCM setups. More importantly, the proposed solution still provides the same effective capacity as the MLC-only PCM at all times.

CHAPTER 6

SUMMARY

Rapidly escalating transistor integration densities have accentuated the perennial divergence between processor and memory performance. The "memory wall" phenomenon hinders the performance gains that may be reaped from the abundance of on-chip computational resources. Consequently, the design of the memory hierarchy and sub-systems has garnered special attention from computer architects over the last several years. Despite the steady increase in the size of on-chip and off-chip memory, the latter can still benefit from even larger capacities. An efficient way to increase the *effective* memory capacity without increasing the *physical* size is memory compression.

In this thesis, Effective Capacity Maximizer (ECM) was proposed as a way to maximize the performance of compressed caches. In a compressed cache, the cacheline size varies depending on the achieved compression ratio. This compressed cacheline size information gives useful hints when managing the cache (e.g., when selecting a victim), which can lead to increased cache performance. Specifically, ECM showed an average effective capacity increase of 18.4% over the Least-Recently Used (LRU) policy, and 23.9% over the Dynamic Re-Reference Interval Prediction (DRRIP) scheme. This increase in effective cache capacity translated into average system performance improvements of 8.7% over LRU and 5.1% over DRRIP.

The potential increase in effective capacity has led researchers to develop various compressed LLC architectures by designing efficient compression algorithms and compression-aware cache structures. However, blindly applying compression everywhere is not always a good idea, as compression has some overhead designers must take into consideration. Therefore, Hot-cacheline Prediction and Early decompression (HoPE) mechanism was proposed to mitigate one of the most important components of the overhead associated with data compression: the read-hit-decompression latency. The simulations demonstrated that

HoPE can reduce the read-hit decompression penalty in compressed LLCs by more than 60%, when compared to compressed caches using the LRU, DRRIP, and ECM. This significant reduction in the read-hit penalty yielded an average increase in overall system performance of 9.8% over LRU, 6.2% over DRRIP, and 4.6% over ECM, respectively.

The Phase Change Memeory (PCM) is increasingly viewed as an attractive alternative for the memory sub-system of future microprocessor architectures. However, weaknesses of PCM have urged designers to develop various supporting architectural techniques to aid and complement the operation of the PCM. One promising such solution is the deployment of hybridized memory architectures that fuse DRAM and PCM, in order to combine the best attributes of each technology. In this thesis, a novel Dual-Phase Compression (DPC) scheme and its architectural design was proposed aimed at DRAM/PCM hybrids. Furthermore, the proposed architecture was imbued with a multi-faceted wear-leveling technique to enhance the durability and prolong the lifetime of the PCM. The proposed DPC scheme showed 27.8% performance improvement and 30.5% energy reduction, on average, as compared to a baseline DRAM/PCM hybrid implementation. Additionally, the multi-faceted wear-leveling technique was shown to significantly prolong the lifetime of the PCM.

The storage density of PCM has been demonstrated to double through the employment of Multi-Level Cell (MLC) PCM arrays. However, this increase in capacity comes at the expense of increased latency (both read and write) and decreased long-term endurance, as compared to the more conventional Single-Level Cell (SLC) PCM. These negative traits of MLCs detract from the potentially invaluable storage benefits. Therefore, a compression-based hybrid MLC/SLC PCM management technique was proposed that aims to combine the performance edge of SLCs with the higher capacity of MLCs in a hybrid environment. The simulations demonstrated that the proposed technique achieves 3.6 times performance enhancement and 72% energy reduction, on average, as compared with MLC-only configurations, while always providing the same effective capacity as the MLC-only mode.

REFERENCES

[1] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pp. 60–71, 2010.

[2] C. Bienia, *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[3] X. Dong and Y. Xie, "Adams: Adaptive MLC/SLC phase-change memory design for file storage," in *Proceedings of the 16th Asia and South Pacific Design Automation Conference*, pp. 31–36, 2011.

[4] S. Baek, H. G. Lee, C. Nicopoulos, J. Lee, and J. Kim, "ECM: Effective capacity maximizer for high-performance compressed caching," in *High-Performance Computer Architecture, 2013. HPCA-19. 19th International Symposium on*, 2013.

[5] A. R. Alameldeen and D. A. Wood, "Adaptive cache compression for high-performance processors," in *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pp. 212–223, 2004.

[6] International Technology Roadmap for Semiconductors, Semiconductor Industry Association, 2010.

[7] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression: A significance-based compression scheme for l2 caches," in *Technical Report 1500*, 2004.

[8] X. Chen, L. Yang, R. Dick, L. Shang, and H. Lekatsas, "C-pack: A high-performance microprocessor cache compression algorithm," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, no. 8, pp. 1196 –1208, 2010.

[9] P. Franaszek, J. Robinson, and J. Thomas, "Parallel compression with cooperative dictionary construction," in *Proceedings of the Conference on Data Compression*, DCC '96, pp. 200–209, 1996.

[10] J.-S. Lee, W.-K. Hong, and S.-D. Kim, "An on-chip cache compression technique to reduce decompression overhead and design complexity," *J. Syst. Archit.*, vol. 46, no. 15, pp. 1365–1382, 2000.

[11] L. Villa, M. Zhang, and K. Asanović, "Dynamic zero compression for cache energy reduction," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pp. 214–220, 2000.

[12] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, MICRO 33*, pp. 258–265, 2000.

[13] A.-R. Adl-Tabatabai, A. M. Ghuloum, and S. O. Kanaujia, "Compression in cache design," in *Proceedings of the 21st annual international conference on Supercomputing*, ICS '07, pp. 190–201, 2007.

[14] E. Hallnor and S. Reinhardt, "A unified compressed memory hierarchy," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pp. 201 – 212, 2005.

[15] E. G. Hallnor and S. K. Reinhardt, "A compressed memory hierarchy using an indirect index cache," in *Proceedings of the 3rd workshop on Memory performance issues: in conjunction with the 31st international symposium on computer architecture*, WMPI '04, pp. 9–15, 2004.

[16] S. Kim, J. Lee, J. Kim, and S. Hong, "Residue cache: a low-energy low-area l2 cache architecture via compression and partial hits," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pp. 420–429, 2011.

[17] Y. Xie and G. Loh, "Thread-aware dynamic shared cache compression in multi-core processors," in *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pp. 135 –141, 2011.

[18] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pp. 377–388, ACM, 2012.

[19] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM Journal of Research and Development*, vol. 52, no. 4/5, pp. 439 – 447, 2008.

[20] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pp. 2–13, 2009.

[21] S. Cho and H. Lee, "Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *International Symposium on Microarchitecture*, pp. 347–357, 2009.

[22] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pp. 14–23, 2009.

[23] A. Bivens, P. Dube, M. Franceschini, J. Karidis, L. Lastras, and M. Tsao, "Architectural design for next generation heterogeneous memory systems," *IEEE International Memory Workshop*, pp. 1 – 4, 2010.

[24] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pp. 24–33, 2009.

[25] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: a hybrid PRAM and DRAM main memory system," in *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pp. 664–469, 2009.

[26] F. Bedeschi and et al., "A bipolar-selected phase change memory featuring multi-level cell storage," *IEEE J Solid State Circuits*, vol. 44, no. 1, pp. 217–227, 2009.

[27] D. H. Kang and et al., "Two–bit cell operation in diode–switch phase change memory cells with 90nm technology," in *Symposium on VLSI Technology*, pp. 10–12, 2008.

[28] H. Liu, M. Ferdman, J. Huh, and D. Burger, "Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency," in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pp. 222–233, 2008.

[29] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pp. 381–391, 2007.

[30] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, Jr., and J. Emer, "Ship: signature-based hit predictor for high performance caching," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pp. 430–441, 2011.

[31] Y. Xie and G. H. Loh, "Pipp: Promotion/insertion pseudo-partitioning of multi-core shared caches," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pp. 174–183, ACM, 2009.

[32] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pp. 187–200, USENIX Association, 2004.

[33] T. Johnson and D. Shasha, "2q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pp. 439–450, 1994.

[34] D. Lee, J. Choi, J. H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, "Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Transactions on Computer*, vol. 50, no. 12, pp. 1352–1361, 2001.

[35] N. Megiddo and D. S. Modha, "Arc: A self-tuning, low overhead replacement cache," in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, (Berkeley, CA, USA), pp. 115–130, USENIX Association, 2003.

[36] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, (New York, NY, USA), pp. 134–142, ACM, 1990.

[37] R. Subramanian, Y. Smaragdakis, and G. Loh, "Adaptive caches: Effective shaping of cache behavior to workloads," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pp. 385–396, 2006.

[38] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pp. 91–104, USENIX Association, 2001.

[39] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, Jr., and J. Emer, "Adaptive insertion policies for managing shared caches," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pp. 208–219, 2008.

[40] Wind River Systems. http://www.windriver.com/.

[41] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (gems) toolset," *SIGARCH Computer Architecture News*, vol. 33, 2005.

[42] CACTI 6.5. http://www.hpl.hp.com/research/cacti/.

[43] Samsung Electronics, "DDR2 registered SDRAM module, M393T5160QZA," *Datasheet*.

[44] Apple - OS X Mavericks, 2013. http://www.apple.com/osx/advanced-technologies/.

[45] H. Fang, C. Tong, B. Yao, X. Song, and X. Cheng, "Cachecompress: A novel approach for test data compression with cache for ip embedded cores," in *Proceedings of the 2007 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '07, pp. 509–512, 2007.

[46] K. Patel, E. Macii, and M. Poncino, "Zero clustering: An approach to extend zero compression to instruction caches," in *Proceedings of the 15th ACM Great Lakes Symposium on VLSI*, GLSVLSI '05, 2005.

[47] E. Ahn, S.-M. Yoo, and S.-M. S. Kang, "Effective algorithms for cache-level compression," in *Proceedings of the 11th Great Lakes Symposium on VLSI*, GLSVLSI '01, 2001.

[48] O. Ozturk, G. Chen, M. Kandemir, and I. Kolcu, "Compiler-guided data compression for reducing memory consumption of embedded applications," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ASP-DAC '06, 2006.

[49] I. Chihaia and T. Gross, "An analytical model for software-only main memory compression," in *Proceedings of the 3rd Workshop on Memory Performance Issues: In Conjunction with the 31st International Symposium on Computer Architecture*, WMPI '04, 2004.

[50] J. Dusser, T. Piquet, and A. Seznec, "Zero-content augmented caches," in *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, 2009.

[51] B. Abali, H. Franke, X. Shen, D. E. Poff, and T. B. Smith, "Performance of hardware compressed main memory," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, (Washington, DC, USA), pp. 73–, IEEE Computer Society, 2001.

[52] S. Gao, J. Xu, B. He, B. Choi, and H. Hu, "Pcmlogging: Reducing transaction logging overhead with pcm," in *Proceedings of the 20th ACM International Conference on Information and Knowledge Management*, CIKM '11, (New York, NY, USA), pp. 2401–2404, ACM, 2011.

[53] M. Zhou, Y. Du, B. R. Childers, R. Melhem, and D. Mosse, "Writeback-aware bandwidth partitioning for multi-core systems with pcm," in *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pp. 113–122, IEEE Press, 2013.

[54] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Bit mapping for balanced pcm cell programming," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pp. 428–439, ACM, 2013.

[55] D. Liu, T. Wang, Y. Wang, Z. Qin, and Z. Shao, "A block-level flash memory management scheme for reducing write activities in pcm-based embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pp. 1447–1450, 2012.

[56] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y. M. Wang, "Age-based pcm wear leveling with nearly zero search cost," in *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pp. 453–458, ACM, 2012.

[57] J. Hu, Q. Zhuge, C. J. Xue, W.-C. Tseng, and E. H.-M. Sha, "Software enabled wear-leveling for hybrid pcm main memory on embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, 2013.

[58] A. P. Ferreira, M. Zhou, S. Bock, B. Childers, R. Melhem, and D. Mossé, "Increasing pcm main memory lifetime," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '10, 2010.

[59] Q. Li, L. Jiang, Y. Zhang, Y. He, and C. J. Xue, "Compiler directed write-mode selection for high performance low power volatile pcm," in *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '13, pp. 101–110, 2013.

[60] A. Hay, K. Strauss, T. Sherwood, G. H. Loh, and D. Burger, "Preventing pcm banks from seizing too much power," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pp. 186–195, 2011.

[61] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pp. 14–23, 2009.

[62] J. Chen, Z. Winter, G. Venkataramani, and H. H. Huang, "rpram: Exploring redundancy techniques to improve lifetime of pcm-based main memory," in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 201–202.

[63] J. Yue and Y. Zhu, "Exploiting subarrays inside a bank to improve phase change memory performance," in *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, 2013.

[64] L. Jiang, Y. Zhang, and J. Yang, "Enhancing phase change memory lifetime through fine-grained current regulation and voltage upscaling," in *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, 2011.

[65] M. Zhou, Y. Du, B. Childers, R. Melhem, and D. Mossé, "Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems," *ACM Trans. Archit. Code Optim.*, pp. 53:1–53:21, 2012.

[66] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the International Conference on Supercomputing*, ICS '11, (New York, NY, USA), pp. 85–95, ACM, 2011.

[67] O. Zilberberg, S. Weiss, and S. Toledo, "Phase-change memory: An architectural perspective," *ACM Computing Surveys*, pp. 29:1–29:33, 2013.

[68] Y. Du, M. Zhou, B. Childers, R. Melhem, and D. Mossé, "Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory," *ACM Trans. Archit. Code Optim.*, 2013.

[69] A. Fawibe, J. Sherman, K. Kavi, M. Ignatowski, and D. Mayhew, "New memory organizations for 3d dram and pcms," in *Proceedings of the 25th International Conference on Architecture of Computing Systems*, ARCS'12, pp. 200–211, 2012.

[70] D.-J. Shin, S. K. Park, S. M. Kim, and K. H. Park, "Adaptive page grouping for energy efficiency in hybrid pram-dram main memory," in *Proceedings of the 2012 ACM Research in Applied Computation Symposium*, RACS '12, 2012.

[71] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase changed based memory systems," in *International Symposium on Microarchitecture*, pp. 2–13, 2009.

[72] J. Kong and H. Zhou, "Improving privacy and lifetime of PCM-based main memory," in *International Conference on Dependable Systems and Networks*, pp. 333–342, 2010.

[73] L. Andrė, Barroso, and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, pp. 33 – 37, 2007.

[74] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam, "Phase-change random access memory: a scalable technology," *IBM J. Res. Dev.*, vol. 52, pp. 465–479, 2008.

[75] S. Kang and et al., "A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth," in *International Solid-State Circuits Conference*, pp. 46 – 48, 2012.

[76] J. Yang, Y. Zhang, and R. Gupta, "Frequent value compression in data caches," in *International Symposium on Microarchitecture*, pp. 258–265, 2000.

[77] G. Sun, D. Niu, J. Ouyang, and Y. Xie, "A frequent-value based pram memory architecture," in *Proceedings of the ASP-DAC 2011*, pp. 211–216.

[78] M. Arjomand, A. Jadidi, A. Shafiee, and H. Sarbazi-Azad, "A morphable phase change memory architecture considering frequent zero values," in *International Conference on Computer Design*, pp. 373–380, 2011.

[79] A. R. Alameldeen and D. A. Wood, "Frequent pattern compression:a significance-based compression scheme for l2 caches," in *Techinical Report 1500, Computer Sciences Department, University of Wisconsin-Madison*, 2004.

[80] R. Das., A. K. Mishra, C. Nicopoulous, D. Park, V. Narayanan, R. Iyer, M. S. Yousif, and C. R. Das, "Performancd and power optimazation through data compression in network-on-chip architectures," *International Symposium on High Peformance Computer Architecture*, pp. 215 – 225, 2008.

[81] W. Zhang and T. Li, "Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures," in *International Conference on Parallel Architectures and Compilation Techniques*, pp. 101–112, 2009.

[82] V. K. Kodavalla, "IP gate count estimation methodology during micro-architecture phase," *IP based Electronic System*, 2007.

[83] S. Schechter, G. H. Loh, K. Straus, and D. Burger, "Use ECP, not ECC, for hard failures in resistive memories," *SIGARCH Comput. Archit. News*, vol. 38, pp. 141–152, 2010.

[84] T. Nirschl, J. Philipp, T. Happ, G. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y. Chen, Y. Zhu, R. Bergmann, H. L. Lung, and C. Lam, "Write strategies for 2 and 4-bit multi-level phase-change memory," in *Electron Devices Meeting, 2007. IEDM 2007. IEEE International*, pp. 461–464, 2007.

[85] M. K. Qureshi, M. M. Franceschini, A. Jagmohan, and L. A. Lastras, "Preset: Improving performance of phase change memories by exploiting asymmetry in write times," *SIGARCH Comput. Archit. News*, vol. 40, no. 3, 2012.

[86] L. Jiang, B. Zhao, Y. Zhang, J. Yang, and B. R. Childers, "Improving write operations in mlc phase change memory," in *Proceedings of the 2012 IEEE 18th International Symposium on High-Performance Computer Architecture*, HPCA '12, 2012.

[87] L. Jiang, Y. Zhang, B. R. Childers, and J. Yang, "Fpb: Fine-grained power budgeting to improve write throughput of multi-level cell phase change memory," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[88] L. Jiang, Y. Zhang, and J. Yang, "Er: Elastic reset for low power and long endurance mlc based phase change memory," in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, pp. 39–44, 2012.

[89] D. H. Yoon, J. Chang, R. S. Schreiber, and N. P. Jouppi, "Practical nonvolatile multilevel-cell phase change memory," in *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, 2013.

[90] N. H. Seong, S. Yeo, and H.-H. S. Lee, "Tri-level-cell phase change memory: Toward an efficient and reliable memory system," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, 2013.

[91] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montaño, and J. P. Karidis, "Morphable memory system: a robust architecture for exploiting multi-level phase change memories," in *Proceedings of the 37th annual international symposium on Computer architecture*, pp. 153–162, 2010.

[92] S. Kang and et al., "A 0.1/spl mu/m 1.8V 256Mb 66MHz synchronous burst PRAM," in *International Solid-State Circuits Conference*, pp. 487 – 496, 2006.

[93] H. Chung and et al., "A 58nm 1.8V 1Gb PRAM with 6.4MB/s program BW," in *International Solid-State Circuits Conference*, pp. 500 – 502, 2011.